Mentor®
A Siemens Business

FormalPro Reference Manual

Release 2018.1

May 2018

Note - Viewing PDF files within a web browser causes some links not to function (see MG595892).
Use HTML for full navigation.

Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: mentor.com
Support Center: support.mentor.com

Send Feedback on Documentation: support.mentor.com/doc_feedback_form

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

FormalPro is a formal verification tool that checks the functional equivalence of two designs. FormalPro operates on the designs using optional user-defined input files, which prepare the designs for matching ports and registers, identifying targets between the two designs, and solving the targets to verify the functional equivalence.

Command Line Syntax Conventions

This document uses notational elements to describe command line syntax.

Bold	A bold font indicates a required argument.
[ ]	Square brackets enclose optional arguments (in command line syntax only). Do not enter the brackets.
Italic	An italic font indicates a user-supplied argument.
underlined	An underlined item indicates either the default argument or the default value of an argument.
…	An ellipsis follows an argument that may appear more than once. Do not include the ellipsis in commands.
{ }	Braces enclose arguments to show grouping. Do not enter the braces.
|	The vertical bar indicates an either/or choice between items. Do not include the bar in the command.

Related Publications

Several types of documents make up the FormalPro document set.

- FormalPro User's Manual — provides process, concept, and procedure information for FormalPro.
- FormalPro Release Notes — provides release information that reflects changes to FormalPro for the software version release.

- Getting Started with FormalPro — offers a tutorial that allows you to use some of the most important aspects of and uses for FormalPro.

Chapter 2
Command Reference

The *formalpro* command is a Linux shell command that invokes FormalPro. You can use it in a shell script or a do file to setup and automate the equivalence checking analysis, or you can use it to invoke the GUI to setup and run the analysis.

For more information on using the GUI and creating shell scripts, see the FormalPro User's Manual.

formalpro Command

Invokes the FormalPro equivalence checker.

Usage

formalpro [*globalOptions*]
{-a *library_specification module_specification design_scope*}
{-b *library_specification module_specification design_scope*}
[-common *library_specification module_specification*]

formalpro -gui [*globalOptions*]

formalpro {-archive | -reports | -verilogFile}

formalpro -help [blackbox | match | constraints | rules]

Arguments

- ***library_specification***
  [-v *libraryFile*] [-y *libraryDirectory*] [-vlibF | -ylibF *dirList*]
  [-alib *libFile* | -alibF *libList*] [-slib *libFile* | -slibF *libList*] [-[no]LibertyPGpins]
- ***module_specification***
  [-mod *moduleName*]
- ***design_scope***

  Design Specification Switches:
  {designFile … | -fl *designFileList*} [-gate | -rtl]

Library Resolution Switches:
[+liborder | +librescan] [+libext+*extension*[+*extension*…]] [+libVerbose] [+noLibCell]

File Language-Specific Switches:
[-87 | -93 | -2008] [-vhdlFile *filename*] [-vhdl2008File *filename*] [-sv *filename...*]
[-svFile *filename*] [-sv2005 *filename*] [-sv2005File *filename*] [-sv2009 *filename...*]
[-sv2009File *filename*] [-verilogFile *filename*] [-vlog95] [-vlog01]
[-edifFile *filename*] [-31aCompat] [-CommonCUnitScope] [-vcsCompat]

Library Mapping Switches:
[-work libraryName { [*fileType*] filePathname... | -fl *designFile*List}]
[-vmapfile filename] [-noGateOptimization name=value]

Compiler Switches:
[-inferVHDLorder] [-synopsysStrictArrayAddress] [-partialSumCheck]
[-dffWithEnable] [+noLibCell] [-paLib<pa_type>] [-pruneMuxAheadOfLatch]
[-optimizeEqOpers] [-rtlTreatDeclAsassign] [-fastVerilogRead]

Power Aware Switches:
[-PACheck] [-paConfigFile] [-paLib<pa_type>] [-paConfig<pa_type>] [-upf]

Verilog-specific Switches:
[+incdir+*include_dir* …] [+define+definition[=value]…] [-parameters name=value]

Control Switches:
[-propagateDontCare {all | none}] [-rtlMemoryLimit [integer]] [-treatDivisionAsShift]
[RTL Naming Control] [-FSMencoding scheme] [-encapsulateAll] [-verifyTristate]
[-commentSynthOffRegions] [-commentTransOffRegions]

Formal Eyes Switches:
[-formalEyesAll] [-formalEyesFloat] [-formalEyesMulti] [-formalEyesX]
[-formalEyesConstRegs]

- **globalOptions**

  Interface Control:
  [-noOverWrite] [-f *command_file*] [-flow {filename | predefinedFlow}]
  [-cache [*cacheDir*]] [-memLimit *number*] [-debug] [-cycleCountLimit *integer*]
  [{-suffixVerilog | -suffixVHDL | -suffixSystemVerilog} *extensionList*]
  [{-suffixVlogLib | -suffixDftLib | -suffixEDIF | -suffixSynLib} *extensionList*]
  [-eco [generate | final]]

  Environment Settings:
  {[-mp *integer*] [-mpLimit *integer*] [-mpTimeLimit *integer*]} [-queueLicense]
  [-noCheckResources]

  Input-file Specification:
  [-blackboxFile *filename*] [-matchFile *filenames*] [-ruleFile *filename*]
  [-addRuleFile *filename*] [-constraintFile *filenames*] [-configFile *filename*]
  [-paConfigFile *filename*]

Flow Control:
[-stopAfter{compile | match}] [-stopOnBlackBox] [-stopOnMissing]
[-stopOnUnmatched] [-stopOnDiff *integer*] [-stopOnConfigError]
[-stopOnConstraintError] [-stopOnCycles] [-resume]
[-restart {a | b | match | constraint | solve | coverage}]

Log-file Control:
[-log *logFileName*] [-logLevel {mini | compact | full}] [-suppress]
[-reportUnmatchedDiffs]

Match-stage Control:
[-convertFloats {floating | 0 | 1 | input | X}] [-bufifenable] [-checkArrayOffsets]
[-diffOnQ] [-diffOnQOnly] [-gatedClocks] [-noheuristicNameLookup]
[-ignoreNoPath] [-matchseq engine[:engine]…] [-mergeReplicatedReg]
[-PACheck {none | all | isolation | level_shifter | retention}] [-removeIgnoredOutputs]
[-simplifyPipelineRegs] [-useAliasPhases] [-rtlIgnoreNoPathBBIns]

Solve-stage Control:
[-solveFedByUnmatched] [-strategy *engine_level*] [-cycleSolve]
[-solveTimeLimit] [-solveOrder *factory_supplied_controlFile*] [-retime]
[-dataPath] [-dataPathModules *mapList*]

Operator Control:
[-multiplierArchitecture *architecture*[_adderType][_swap]]
[-dividerArchitecture *architecture*] [-log]

Library Modeling:
[-QQbarMerge] [-QQbarSetResetMerge] [-masterSlaveMerge]
[-redundantRegMerge] [-libConfigFile *configFileName*]

FPGA Control:
[{-fvi *file*.fvi | -wsp f*ile*.wsp}]

-fpga {altera | actel | xilinx}

Note: *the -fpga option is only required when you run FormalPro using the command invocation* formalpro_fpga.

Description

FormalPro verifies the functional equivalence of two designs, either gate-level netlists or Register-transfer level (RTL) design files. FormalPro produces various reports and comparison data you can use to determine the source of any differences found between the designs.

You can invoke FormalPro from the command line, or through the Graphical User Interface (GUI), which you invoke by using the **-gui** switch.

Specify the design files to be verified within the design scope (identified by the -a and -b switches). It is a common practice to specify design A as your reference design and design B as

your altered design. The design scope is also where you specify any design specific switches. Specify common library files for both designs in the -common design scope.

The *globalOptions* group of arguments controls the interface and process flow of the run. You should specify the global options before the -a design scope.

As FormalPro runs, it creates a cache that stores information generated at each stage of the run. This cache is normally stored in the current directory as *formalpro.cache/*.

After the run has completed, you should analyze the results, as shown in the log file, to verify that every stage of the flow completed as you expected. If so, and FormalPro reports there are differences, use the FormalPro debugger to pinpoint the differences. For information on the debugger, refer to the chapter, "Debugging Design Differences" in the *FormalPro User's Manual* or see "Debugger Commands" in this manual.

Examples

The following example verifies two Verilog designs, *lpfir.v* and *lpfir_scan.v*, containing technology cells from the same library, *../lib/js2bp.v*:

```
formalpro -a lpfir.v \
      -b lpfir_scan.v \
      -common -v ../lib/js2bp.v
```

The following example compares the top-level module of the Verilog files in the directory *preTweak/* with the module DUM in *postTweak.v* using the ATPG library *asic_lib.lib* for both designs. The comparison stops if any black box, unmatched item, or difference is encountered:

```
formalpro -stopOnBB -stopOnUnmatched -stopOnDiff 1\
      -a preTweak/*.v \
      -b postTweak.v -mod DUM \
      -common -alib asic_lib.lib \
```

The following example compares *original.v* in the *vendor1* Verilog library directory and *retarget.v* in the *vendor2.v* library, with an additional matching file called *matches.cmd*.

```
formalpro -matchFile matches.cmd \
   -a original.v -y vendor1/ \
   -b retarget.v -v vendor2.v
```

formalpro Command Options

The *formalpro* command provides switch options to support a variety of operations.

-31aCompat

Scope: Design-specific

Alias: none

Reduces compilation errors generated by older SystemVerilog files.

Usage

- -31aCompat — Reduces errors generated by legacy SystemVerilog files when compiled by the RTL compiler.

Description

Legacy SystemVerilog files that conform to the LRM version 3.1A may not be compliant with later versions of SystemVerilog. This option eases the restrictions imposed by the RTL compiler in this case.

GUI Access

Location:	**Options** dialog box —
	A specific pane > **RTL** tab
	B specific pane > **RTL** tab
Action:	Enable: select **Version 3.1A compatibility**
	Disable: deselect **Version 3.1A compatibility**

Examples

```
formalpro -a test.sv mydff.v -31acompat -b netlist.v
```

-87 | -93 | -2008

Scope: Design-specific

Alias: None

Specifies the version of subsequent VHDL design files.

Usage

- -87 — VHDL87
- -93 — VHDL93 (default)
- -2008 — VHDL2008

Description

This switch must precede design files to which it applies. Switches apply to all subsequent VHDL design files within the design a scope unless another switch is specified. These switches can be used on a file-by-file basis.

By default (when FormalPro is first installed), all files with the extension vhd, .VHD, .hdl, .HDL, .vhdl, and .VHDL are compiled as VHDL files based on the settings in formalpro.ini. This behavior can be changed by changing the formalpro.ini settings or using one of the suffix control switches.

GUI Access

Location:	**Project** tab > **A** tab
	Project tab > **B** tab
Action:	Select **87**, **93**, or **2008** from the dropdown menu to the left of the specific design file.

Examples

In the following example design A consists of 3 design files, where two are in VHDL87 format and one is in VHDL93 format. For design B, the default behavior is to read the design file as VHDL93 format.

```
formalpro -a -87 design_1a.vhd design_1b.vhd \
            -93 design_1c.vhd \
          -b design_2.vhd
```

Related Topics

Suffix Control Switches (Library Files)

Suffix Control Switches (Design Files)

-vhdl2008File

-vhdlFile

-a, -b, -common

Scope: Design-specific

Alias: None

Specifies the beginning of a design scope.

Usage

- -a — Specifies the beginning of design A, typically the reference design.
- -b — Specifies the beginning of design B, typically the modified design.
- -common — Specifies the beginning of the common design scope, where you specify the top-level module for and/or libraries that apply to both designs A and B.

Description

The design scope of the FormalPro command line is where you specify any switches that are specific to either the A or B design. The reference table in the upper right-hand corner of each switch's reference page shows whether a switch is design-specific or not.

GUI Access

Location:	**Project** tab > **A** tab
	Project tab > **B** tab
	Project tab > **Common** tab

Examples

```
formalpro -a reference_design.v \
          -b modified_design.v \
          -common -v shared_libraries.v
```

-addRuleFile

Scope: Global

Alias: None

Specifies the location of an additional rule file containing implicit match rules.

Usage

- -addRuleFile filename

filename — Specifies the location of a rule file. Non-literal pathnames are relative to the current directory.

Description

FormalPro automatically loads a default rule file that contains several pre-created rules that aid in matching the comparison points between your designs. You can use this option to specify an additional rules file (or from the file specified by the -ruleFile command option).

Note

Before using this option you should have a full understanding of the operation and interaction of the default rule file, rule sets, the -ruleFile option and the -addRule file option. For this information, see the section titled "Rule Files" on page 207.

GUI Access

Location:	**Project** tab > **General** tab > **Match rules** entry box
Action:	Type the path to your rule file or use: 

Examples

```
formalpro -addrulefile ./setup/my_rule.cmd \
          -a design_a.v \
          -b design_b.v
```

-alib, -alibF

Scope: Design-specific

Alias: None

Specifies Mentor Graphics ATPG library files to use for design compilation.

Usage

- -alib *libFile* —Specifies an ATPG technology library. One *libFile* argument per switch.
- -alibF *libList* — Specifies a file containing a list of ATPG technology library files. One libList argument per switch.

Description

Use these options for specifying a library file or a list of library files for design compilation.

You can specify -alib and -alibF multiple times on the command line.

The format of *libList* is shown in the following example:

```
# commented line
./lib/atpg_library_1.lib          # a single ATPG library file
./lib/atpg_lib_*.lib              # wildcards are allowed
```

GUI Access

Location: **Project** tab > **A** tab
Project tab > **B** tab
Project tab > **Common** tab

Action: Specifying a library file:

1. Type the path to your library or use: 
2. Select **alib** from the dropdown menu to the left of the library.

Specifying a list of library files:

1. Type the path to your file list or use: 
2. Select **alibF** from the dropdown menu to the left of the file.

Examples

```
formalpro -a designA.v -b designB.v \
          -common -alib ./atpg_lib_1.atpglib
```

-archive

Scope: Stand-alone

Alias: None

Archives a FormalPro cache.

Usage

- -archive — Saves the cache containing results of a previous run. By default, subsequent runs overwrite the existing cache.

Description

The cache is renamed to formalpro.cache_archive_<n>, where n incrementally increases beginning at 1, and retains the subdirectories debug, logs, inputFiles, outputFiles, and reports. The -restart switch cannot be used on the archived data.

GUI Access

Location: **File > Archive cache**

Examples

The following command:

```
formalpro -archive
```

creates an archive of the current formalpro.cache as the following directory:

```
formalpro.cache_archive_1
```

-blackboxFile

Scope: Global

Alias: -bbfile

Specifies the location of a black box file containing user-created black box definitions.

Usage

- -blackboxFile *filename*

filename — Specifies the location of the black box file. Non-literal pathnames are relative to the current directory.

- -noblackboxFile *filename*

filename — Specifies a black box file to ignore when restarting a previous run.

For more information about black box files, see “blackbox, encapsulate, and noencapsulate” on page 216.

GUI Access

Location:	**Project** tab > **General** tab > **Black box** entry box
Action:	Type the path to your black box file or use: 

Examples

```
formalpro -blackboxfile ./setup/bb.cmd \
          -a design_a.v \
          -b design_b.v
```

-bufifenable

Scope: Global

Alias: None

Enables comparison points on tri-state devices.

Note

This option has been deprecated. Use -verifyTristate.

Usage

- -bufifenable — Adds a comparison point to enable lines on tri-state devices if the tri-state bus feeds either a primary output port or the input to a blackbox.
- -nobufifenable — Disables this functionality (default).

Description

In order to address a design scenario in which one design is driving a Z value on a bus when the other design is driving a 0, FormalPro provides a comparison point for the enable lines on tri-state devices. Tri-state devices are those that are described by the Verilog primitives bufif0, bufif1, notif0, and notif1. If a tri-state bus is driven by several tri-state devices, the enable lines are logically ORed together. The output of the OR is matched to a corresponding OR in the other design and treated as a primary output. This enable signal will have the same name as the tri-state bus with the string "_MGC__BUFIF_ENABLES" appended.

Since the enable-signal name is processed using the normal name matching methods, it may be necessary to manually match this comparison point if the names of the two designs are significantly different.

This comparison point is only added if the tri-state bus feeds either a primary output port or the input to a blackbox. This limitation results from the likelihood that the interior logic of two designs will be different and that there won't be a reasonable match between the A and B sides. Primary outputs and blackbox inputs are much more likely to have a correspondence between the two designs.

GUI Access

Location:	**Options > General > Solve**
Action:	Enable: (default) Select **Solve Bufif Enable**
	Disable: Unselect **Solve Bufif Enable**

Examples

```
formalpro -bufifenable
```

-cache

Scope: Design-specific

Alias: None

Writes the FormalPro cache to a user-specified location.

Usage

- -cache *cacheDir*

cacheDir — Specifies a directory in which to write the FormalPro cache.

Description

FormalPro creates the new cache directory if none exists. If a cache already exists at the specified location, it is overwritten and the run proceeds.

The default location for the FormalPro cache is ./formalpro.cache.

GUI Access

Location:	**Project tab > General tab > Cache directory**
Action:	Type the path of the new FormalPro cache or use: 

Examples

```
formalpro -cache ./new_formalpro.cache \
          -a design_a.v \
          -b design_b.v
```

-checkArrayOffsets

Scope: Global

Alias: None

Enables the name-matching algorithm to use offset to account for port and register arrays that do not begin with the least significant bit (LSB) set to 0.

Usage

- -checkArrayOffsets — Enables this functionality.
- -nocheckArrayOffsets — Disables this functionality (default).

Description

-checkArrayOffsets checks for port and register arrays that do not begin with the least significant bit (LSB) set to 0 and adjusts the name-matching algorithm to account for this offset.

By default, name matching is based on strict name-matching strings, which has the potential to generate mismatched comparison points.

For example, given the following register declaration in Verilog for design A:

```
reg [0:5] regArray ;
```

and the register declaration in Verilog for design B:

```
reg [1:6] regArray;
```

By default, name matching uses strict matching rules that leave registers *regArray[0]* from design A and r*egArray[6]* from design B unmatched, while incorrectly matching *regArray[1:5]*. With the **-checkArrayOffsets** option**,** FormalPro, examines the array offsets and use name matching to match design A, *regArray[0]* to design B, *regArray[1]* and so on.

GUI Access

Location:	**Options > General > Match**
Action:	Enable: (default) Select **checkArrayOffsets**
	Disable: Unselect **checkArrayOffsets**

Examples

```
formalpro -checkArrayOffsets \
```

-noCheckResources

Scope: Global option

Alias: None

Disables available memory checking prior to verification.

Usage

- -checkResources — Checks available system memory and outputs a warning message if there is less than 10 GB of disk space or 2 GB of physical RAM (default).
- -noCheckResources — Disables memory check.

Description

The default behavior can also be changed in the *FORMALPRO_HOME/lib/formalpro.ini* file or the *.wsp* file by setting the *checkResources* option to false.

Examples

The following command disables the memory check by setting the *checkresources* variable in the .wsp file to false:

```
formalpro -wsp file.wsp -noCheckResouces
```

A warning message similar to the following displays if the available disk space is below10 GB:

```
Warning: Low disk space for formalpro.cache 9876K blocks
```

A warning message similar to the following displays if the available RAM is below 2 GB:

```
Warning: Low physical RAM available  1900K blocks
```

-commentSynthOffRegions

Scope: Design-specific

Alias: None

Treats the region between pragmas *synthesis_off* and *synthesis_on* as comments.

Usage

- -commentSynthOffRegions — Enables this functionality.
- -nocommentSynthOffRegions — Disables this functionality (default).

Description

This command instructs FormalPro to treat the region between the pragmas synthesis_off and synthesis_on as comments. This is not default synthesizer behavior.

-commentTransOffRegions

Scope: Design-specific

Alias: None

Treats the region between *synthesis_off* and *synthesis_on* pragmas as comments.

Usage

- -commentTransOffRegions —Treats the region between the pragmas *translate_off* and *translate_on* as comments. This is not default synthesizer behavior.
- -nocommentTransOffRegions — disables this functionality (default).

-CommonCUnitScope

Scope: Design-specific

Alias: none

Compiles design scope files as common unit.

Usage

- -CommonCUnitScope — Specifies that all the files in the design scope (-a or -b) should be compiled as a common compilation unit.

GUI Access

Location:	**Options** dialog box —
	A specific pane > **RTL** tab
	B specific pane > **RTL** tab
Action:	Enable: select **Common compilation unit scope**
	Disable: deselect **Common compilation unit scope**

Examples

In this example, the file1.v, file2.sv and mydiff.v will be compiled in one compilation unit instead of three individual compilation scopes.

```
formalpro \
-a -commoncunitscope file1.v file2.sv mydff.v\
-b netlist.v
```

-configFile

Scope: Global

Alias: None

Specifies the location of configuration files containing user-created constraint statements that are used in the FPGA flows.

Usage

- -configFile *filename* — Specifies the location of the configuration file. Non-literal pathnames are relative to the current directory.
- -noconfigFile — Instructs FormalPro to ignore a specified configuration file when restarting a previous run.

For more information, see "Constraint Files" on page 220.

GUI Access

Location:	**Project** tab > **General** tab > **Config file** entry box
Action:	Type the path to your configuration file or use: 

Examples

```
formalpro -configfile ./setup/config.cmd \
          -a design_a.v \
          -b design_b.v
```

-constraintFile

Scope: Global

Alias: None

Specifies the location of a constraint file containing constraint statements.

Usage

- -constraintFile *filename*

filename — Specifies the location of the constraint file. Non-literal pathnames are relative to the current directory.

- -noconstraintFile — Instructs FormalPro to ignore a specified constraint file when restarting a previous run.

For more information, see "Constraint Files" on page 220.

GUI Access

Location:	**Project tab > General tab > Constraints entry box**
Action:	Type the path to your constraint file or use: 

Examples

```
formalpro -constraintfile ./setup/formalpro.constraint \
          -a design_a.v \
          -b design_b.v
```

-convertFloats

Scope: Global

Alias: None

Converts floating nets to a logical value or treats them as a primary input.

Usage

- -convertFloats [floating | 0 | 1 | input | X]
  - 0 — Sets floating nets to the value “0”.
  - 1 — Sets floating nets to the value “1”.
  - input — Treats floating nets as primary inputs.
  - floating — Treats nets as floating, overriding a previous setting (default). See example.
  - X — Specifies don’t care.

Description

FormalPro applies this switch during the match stage when it is propagating all constants.

FormalPro, by default, does not alter a floating net in your design. It reports all floating nets in the formalpro.cache/reports/floating.report file. You can access the floating nets report from the Reports menu.

Any net in your design without a driver is considered a floating net, and will not be solved. You could use the “0” or “1” arguments to force floating nets to known values, which is useful in the case of verifying an incomplete design. For more information, see Floating Nets in the *FormalPro User’s Manual*.

GUI Access

Location: **Options** dialog box —
General pane > **Match** tab > **Floating nets** dropdown box.

Action: Select argument.

Examples

In this example, all floating nets are converted to primary inputs.

```
formalpro -convertfloats input \
          -a design_a.v -b design_b.v
```

In an FPGA flow (see Verifying FPGA Designs in the *FormalPro User's Guide*), a *design_name*.flow file is generated that sets floating nets to "0" (-convertFloats 0). If you want floating nets to be treating as floating, invoke the .wsp file as follows:

```
design_name.wsp -convertFloats floating
```

-cycleCountLimit

Scope: Global

Alias: None

Controls the number of combinational cycles are reported in the cycles.report file.

Usage

- -cycleCountLimit *integer*

integer — Specifies the maximum number of combinational cycles reported in the cycles.report file. Default is 100.

Description

FormalPro reports all combinational cycles, including all paths to an output from the cycle, to the formalpro.cache/reports/cycles.report file.

From the GUI, you can access the combinational cycles report from the Reports menu.

GUI Access

Location: **Options** dialog box —
General pane > **Solve** tab > **Cycle count limit** entry box

Action: Enter the maximum number of cycles that should be reported.

Examples

The following example changes the maximum number of cycles that FormalPro reports.

```
formalpro -cyclecountlimit 10 \
          -a design_a.v \
          -b design_b.v
```

-cycleSolve

Global

Alias: None

Disables the solve strategy that proves the equivalence of targets fed by combinational feedback network topologies (cycles).

Usage

- -cycleSolve — enables this functionality (default).
- -nocycleSolve — disables this functionality.

Description

This functionality may increase the runtime for very large designs that contain a lot of combinational cycles. Specify -nocycleSolve to analyze any cycles in your design after the verification run completes.

```
Number of Equivalent comparison points: 232
-----------------------------------------------------
   Solved combinational Cycle 20
```

The tool reports the number of targets solved using this strategy in the Comparison Summary section of the FormalPro log file as follows:

You can access a report on combinational cycles from the **Reports > Removed > Combinational Cycles**.

In some cases, FormalPro may not be able to solve a target fed by a combinational cycle. In this case, it is removed from the verification and reported in the log file as Removed: fed by a combinational cycle. For more information, see Combinational Feedback Loops in the *FormalPro User's Manual*.

GUI Access

Location:	**Options** dialog box — **General** pane > **Solve** tab
Action:	Enable: Select **Solve targets with combinational cycles** Disable: Unselect **Solve targets with combinational cycles**

Examples

```
formalpro -nocyclesolve \
          -a design_a.v \
          -b design_b.v
```

-dataPath

Global

Alias: None

Enables the FormalPro datapath solver.

Usage

- -dataPath — Enables the function.
- -noDataPath — Disables the function. (default)

Description

Activates additional match and solve processes to identify arithmetic operators in the system and compare RTL-to-gates with advanced methods. This option can enhance runtime performance considerably when it is applicable. This option does not apply to gate-to-gate compares. When the -dataPath option is active, additional compare points are created at the datapath-related module boundaries. The compare points are of the User defined output (Uout) type in the main FormalPro logs. The Uout is named as an instance and appears as a "+" symbol on the design schematics. A datapath_objects.report report file is generated in the *cache/reports* directory with the instance names of the boundary compare items. If a certain data path collection of operators is not automatically found and processed, you can aid the process in two ways:

1. Provide a user match file that pairs items from the datapath_objects.report to each other. The syntax for a manual match of datapath objects is as follows:

   ```
   match register \A.instance \B.instance
   ```

2. Provide a manual datapath grouping command in a black box file. See the dpAddGroup command.

When datapaths are identified and matched from A to B, the *solve.log* in *cache/logs* will indicate the successful compares related to datapaths. In the following log, two new engines are indicated as DFG and ISODP. The points removed by ISODP are likely to be the boundary points of unused datapaths. These points were potential compares but were not necessarily needed and are benign when unused. If user matches enhanced the number of fully matched datapaths, this number would reduce. When a datapath is rejected by the solver, the Uout points that ringed that module are made transparent and the logic cone at the designation register sees the full path between classic compare points.

```
BEGIN: solve       [11/8/2017 21:09:31]
Solve
================================================================================
Engine   Targets    Equiv.    Diff.    Unsolv.  Removed/   Elapsed        %
 run       fed     targets   targets   targets  Deferred   hh:mm:ss     Compl.
--------------------------------------------------------------------------------
ISO        18294      4956         0     13338         0    0:00:22      27.09
DWC        13338      4527         0      8811         0    0:02:23      51.84
DFG         8811       253         0      8558         0    0:00:08      53.22
ISODP       8558         1         0      7776       781    0:00:05      57.49
DWC         7776        90         0      7686         0    0:00:50      57.99
OMEGA       7686         0         0      4669      3017    0:00:36      74.48
RSYN        4669         3         0      4666         0    0:00:33      74.49
CPP_1       4666      3731         0       935         0    0:21:36      94.89
ERGO         935        95         0       840         0    0:10:57      95.41
ISO          840         0         0       840         0    0:00:03      95.41
RSYN         840         0         0       840         0    0:02:20      95.41
Solve Time Limit Expired.
BenignDP    -781         0         0         0      -781   00:00:00
Deferred              2992         0       +25     -3017   00:00:00      95.06
--------------------------------------------------------------------------------
Totals     17513     16648         0       865         0    0:39:53      95.06
END:   solve       [11/8/2017 21:49:33]  Result: OK
```

GUI Access

Location: **Options** dialog box —
General pane > **Solve** tab

Action: Check the box to enable the datapath solver

Examples

Add the option -datapath to the command line

```
formalpro -a -rtl rtl.f -b -gate gates.f -common -slibf libs.f -datapath
```

Related Topics

-dataPathModules

dpAddGroup

-dataPathModules

Global

Alias: None

Specifies the prefix for the names of datapath objects in the current design.

Usage

- -dataPathModules *mapList* — Specifies a map to name datapath objects. The mapList consists of space delimited pairs of prefix and type in this format:

```
<module_name:module_type ...>
```

Description

This command line or *formalpro.ini* file option can override defaults to specify names of datapath objects in the current design.

The format of the command is the module name paired with a known FormalPro type. Each module_name is considered a naming prefix that must match and can be followed by any string.

The following is a list of the known types of datapath objects:

- mult_sign
- mult_uns
- adder
- incr
- decr
- minus (apply 2’s complement)
- subtract
- datapath

Normally, you do not need to modify this option. Most of the module names are RTLC compiler operators.

GUI Access

None

Examples

The default for this option is shown as an entry in the *formalpro.ini* file:

```
dataPathModules = "M_RTLSIM_MULT_SIGN_:mult_sign
M_RTLSIM_MULT_UNS_:mult_uns M_RTLSIM_ADD_:adder M_RTLSIM_INCR_:incr
M_RTLSIM_DECR_:decr M_RTLSIM_SUB_UNARY_:minus M_RTLSIM_SUB_BIN_:subtract
datapath__:datapath datapath:datapath"
```

The last two module_names in the example are netlist module prefixes.

If, for example, you do not want to process adders with the advanced method, remove the *M_RTLSIM_ADD_:adder* entry from the command. The entire quoted command string is needed for each usage.

Related Topics

-dataPath

dpAddGroup

-debug

Global

Alias: None

Controls the amount of information generated for cross-probing between the FormalPro debug tool and the original design files.

Usage

- -debug — generates cross-probe debug information (default).
- -nodebug — does not generate cross-probe debug information.

Description

The information generated during the Compile stage allows for easier cross-probing from the debug tool back to the original design files.

The -nodebug option limits the generation of cross-probing information.

GUI Access

Location:	**Options** dialog box — **General** pane > **Control** tab
Action:	Enable: **Select Debug info** Disable: **Unselect Debug info**

Examples

```
formalpro -debug \
          -a design_a.v \
          -b design_b.v
```

+define+definition[=value]

Design-specific

Alias: None

Compiles a Verilog definition contained in one of your RTL design files.

Usage

- +define+*definition*[=*value*]...
  - *definition* — Name of the definition.
  - *value* — Values assigned to the definition (optional).

Description

Use this switch if your Verilog RTL files contain ‘ifdef statements, and you want to compile the code within the statement.

GUI Access

Location:	**Project** tab > **A** tab
	Project tab > **B** tab
Action:	Type the switch and argument on a new line, as you would a design file location.

Examples

Assume that a Verilog RTL design design_a.v contains the following code:

```
‘ifdef INCLUDE_REG
   reg [31:0] data_out; ‘endif
```

For FormalPro to properly compile this design, you need to specify that the above code is used, as shown in this example:

```
formalpro -a +define+INCLUDE_REG design_a.v \
          -b design_b.v
```

designFile

Design-specific

Alias: None

Specifies the location of design file(s) to verify.

Usage

- *designFile* — Specifies the files that make up the design to verify.You can specify any number of space-separated files. Wildcards are allowed.

Description

This argument specifies the files that make up the design to verify. You can also specify design files listed within a file with the -fl switch.

FormalPro processes design files in the order they are specified on the command line. If you specify a directory, every file within the directory is used as a design file.

GUI Access

Location:	**Project** tab > **A** tab **Project** tab > **B** tab **Project** tab > **Common** tab
Action:	1. Type the path to your design file or use:  2. Ensure that the dropdown menu to the left of the entry is set to the empty value.

Examples

```
formalpro -a ./design/des_a1.v ./design/des_a2.v \
          -b design_b.v

formalpro -a ./design/des_a*.v \
          -b design_b.v
```

-dffWithEnable

Design-specific

Alias: None

Determines whether D Flip-Flop (DFF) primitives use a clock-enable port.

Usage

- -dffWithEnable (default) — Enables DFF primitives that use a clock-enable port.
- -noDffWithEnable — Disables DFF primitives that use a clock-enable port and instead, creates *enable logic* on the DFF port with the enable-port tied to true.

Description

The default behavior can also be changed in the *FORMALPRO_HOME/lib/formalpro.ini* file or the *.wsp* file by setting the *dffwithenable* option to false.

GUI Access

Location:	**Options** dialog box
	A specific pane > **RTL** tab
	B specific pane > **RTL** tab
Action:	Enable: select **DFF with enable**
	Disable: deselect **DFF with enable**

Examples

```
formalpro -a test.vhd mydff.v -nodffwithenable -b netlist.v
```

-diffOnQ

Global

Alias: None

Disables the creation of comparison points at the Q output of sequential elements in addition to the comparison points at the D, Set, Reset, Enable and Clock inputs.

Usage

- -diffOnQ — Creates comparison points (default).
- -nodiffOnQ — Disables functionality.

Description

FormalPro creates a comparison point for the Q output, in addition to the default comparison points. For an example of this behavior, refer to the heading, “Different Clocking Schemes,” in section “Handling Known Differences in Designs” of the FormalPro User’s Manual.

Tip

When debugging a specified -diffOnQ, debug the Q targets, rather than the inputs of the sequential element.

GUI Access

Location:	**Options** dialog box —
	General pane > **Solve** tab
Action:	Enable: Select **Solve Q targets**
	Disable: Unselect **Solve Q targets**

Examples

```
formalpro -diffonq \
          -a design_a.v \
          -b design_b.v
```

-diffOnQOnly

Global

Alias: None

Disables the creation of comparison points at the Q output of sequential elements without creating any comparison points on the inputs.

Usage

- -diffOnQOnly — Enables functionality (default).
- -nodiffOnQOnly — Disables functionality.

Description

FormalPro creates a comparison point only for the Q output. For more information about comparison points on inputs and outputs, refer to the heading, “Different Clocking Schemes,” in section “Handling Known Differences in Designs” of the FormalPro User’s Manual.

Tip

When debugging a run where you specified **-diffOnQOnly**, you should always debug the Q targets, rather than the inputs of the sequential element.

The logic cones fanning into the Q output target include all of the logic contained in the targets on the inputs of the register. By solving only the Q target, you avoid solving much of the logic multiple times. Not only are there fewer targets to be solved when you use the -diffOnQOnly option, but you also eliminate the “ignorable” differences that are reported on the register inputs; FormalPro is no longer solving the input targets individually. Ignorable differences on the inputs occur when you are verifying gated-clock designs.

GUI Access

Location:	**Options** dialog box —
	General pane > **Solve** tab
Action:	Enable: Select **Solve only Q targets**
	Disable: Unselect **Solve only Q targets**

Examples

```
formalpro -diffonqonly \
          -a design_a.v \
          -b design_b.v
```

-dividerArchitecture

Design-specific

Alias: None

Specifies the architecture used for compiling dividers in your design so that they match the architecture used by your synthesis tool.

Usage

- -dividerArchitecture *architecture*

Architecture— Specifies which architecture to use when compiling dividers. Options include:

- rpl — Ripple adder (default).
- cla — Carry-look-ahead adder.
- cla2 — Carry-look-ahead adder.

Description

This switch specifies that any DW_div divider in an RTL design has been compiled as a specific architecture in your gate-level design. This information instructs the compile stage on how to generate the internal database to most closely resemble your gate-level design. For further information on how FormalPro compiles dividers, refer to the section "Specifying Divider Architectures" in the *FormalPro User's Manual*.

GUI Access

Location: **Options** dialog box —
A specific pane > **RTL** tab >
B specific pane > **RTL** tab >
Divider architecture dropdown menu

Action: Select argument.

Examples

For this example, FormalPro will compile all dividers in your RTL design with the carry-look-ahead architecture.

```
formalpro -a ./rtl/design_a.v -dividerarchitecture cla \
          -b ./gate/design_b.v
```

-DWPipeTransparent

Global

Alias: None

Makes certain DesignWare modules transparent internally with the transparent object command.

Usage

- -DWPipeTransparent — Makes some DesignWare modules transparent.
- -noDWPipeTransparent — Performs full testing for DesignWare modules.

Description

Certain DesignWare modules are always retimed by gate-level synthesis tools. When this option is enabled, the modules will be made transparent by internally using the transparent module command. This user option accounts for all name manipulations and will cover all instances. When a retimed module is made transparent, its registers are changed to buffers and the combinational logic is tested without the false errors from retimed register placement. You must inspect and verify that the registers are sequentially correct for his application. The fp_utility “pipeline_retime” and application note support this effort as do other Mentor products.

The following modules are affected by this option:

- dw_div_pipe

- dw_mult_pipe
- dw_prod_sum_pipe
- dw_sqrt_pipe
- dw02_mult_2_stage
- dw02_mult_3_stage
- dw02_mult_4_stage
- dw02_mult_5_stage
- dw02_mult_6_stage
- mgc_mul_pipe

You can inspect the resulting constraints from the internal text file **.cache/internal/match/outputFiles/{A,B}.SysConstraints*.

-eco

Scope: Global

Alias: None

For designs with functional differences, identify a minimal logic change from the A side (the reference or desired functionality) that can correct the differences if substituted properly into the B side. If requested, the B side design is edited and the modified netlist can be written out. These identified difference regions can also be displayed in the debug schematic to aid in the debug process.

Usage

- -eco generate — Analyze difference regions in the A and B side. Generate a patch module containing Verilog primitives that describe the difference as well as a report to identify characteristics of the difference regions.
- -eco final — Given information created using “-eco generate”, insert patch logic properly into B side modules to correct the differences.

Description

The use of LEC in an Engineering Change Order (ECO) flow can minimize the time needed to modify the original design such that new behavior in the ECO design is seen. The LEC compare feature ensures that the final netlist is a functional match to the required change. For all subsequent features, the user first successfully runs the FormalPro tool to create a cache where logic differences are detected. Using the “-eco generate” option, the tool analyzes the difference region between the A and B side designs. The tool writes out logic changes learned from the A side that correct the related B side modules. These change-logic modules are in terms of

unmapped Verilog primitives and are written out into the *<cache>/eco* directory with names of the form "*unmappedECO_<moduleName>.v*". A report is created and also written out to the *<cache>/reports/ecoGenerate.report* file. The *ecoGenerate.report* file contains information such as the A side primitives used in the patch, the B side primitives and cells that are removed, and the A and B side boundary signal pairs that define inputs and outputs of the patch logic. After the "-eco generate" option completes, generated logic and information can be used in two ways. The first use model is if the user is engaged in an ECO flow where changes are to be applied in a minimally invasive way to an already laid out design, late in the design cycle. The second use model is a user debugging differences encountered in the classic LEC usages.

During an ECO flow, data generated by the "-eco generate" option is helpful in two ways:

- The report file and patch logic can give an indication of the size and scope of the ECO. In some cases, the ECO might be too large in terms of logic changes or too risky in terms of the amount of logic or patch IO which is necessary.
- Assuming the patch logic is acceptable, the user then synthesizes the unmapped change logic into optimized and technology-mapped Verilog modules. After this step, the "-eco final" option is used.

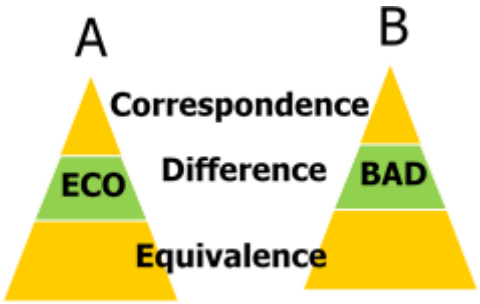
For the user involved in debugging a logic difference, data generated by the "-eco generate" option may be helpful. The FormalPro tool uses significant CPU resources in some cases trying to find the minimal logic differences between the two designs. This can help speed up the debugging task overall in two ways:

- A user might simply examine the report and difference logic to better understand and help debug differences between the two designs.
- If a target schematic is viewed in the debugger for a particular difference target, coloring is used to indicate regions of these target cone(s). These regions are the Equivalence region (white by default), the Difference region (red by default), and the Correspondence region (orange by default).

The Correspondence region is enabled with ECO capabilities. When understanding the logic driving the A and B sides of a target, it is helpful to define these regions:

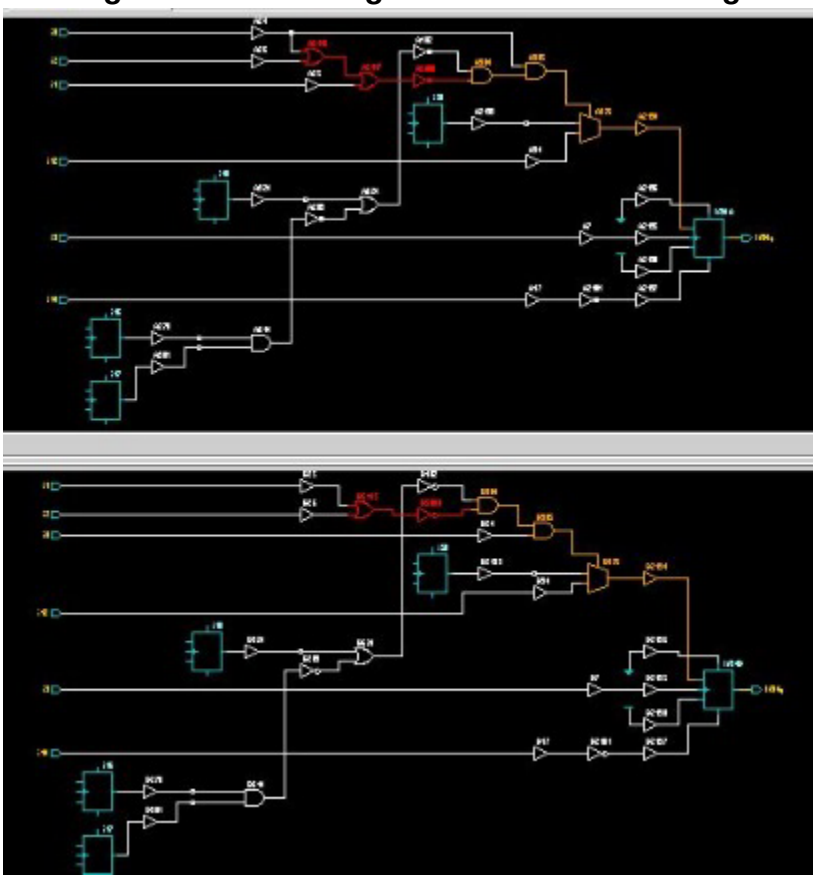
- Equivalence region — The region of target fanin logic that is proven equivalent.
- Difference region — The region of target fanin logic that is proven in-equivalent.
- Correspondence region — The region of target fanin logic that is driven by the difference region but is equivalent. This region is not proven equivalent in the classic sense because it is driven by the difference region which is proven in-equivalent.

These regions are illustrated in the following diagram, which represents A side and B side logic for one target. The objective of the “-eco generate” option is to identify the minimal ECO and bad logic, as well as the input and output signals of that logic.



An example schematic with default coloring for these regions is shown in Figure 2-1. The equivalences are white, the correspondences are orange, and the actual logic differences are red. Change these coloring options in the **GUI Preferences** tab under the **ECO Colors** tab.

Figure 2-1. ECO Region Schematic Coloring



As alluded to earlier, if the user is performing an ECO of a laid-out design, then further steps, as follows, are required after the “-eco generate” option:

- Each unmapped module in the *<cache>/eco* directory must be synthesized (optimized and technology mapped) into a corresponding *mappedECO_<module>.v* file. For example, if a Verilog module is named *modA* then synthesis uses *unmappedECO_modA.v* as input and produces *mappedECO_modA.v* as output. Note that the file name prefixes of *unmappedECO_* and *mappedECO_* are required and the naming scheme for the module name must be followed. The file location must also use the *<cache>/eco* path for correct operation.
- Run the FormalPro tool again using the “-eco final” option. This produces the final “patched” design and modules. For each module in the design with edits (contained in *mappedECO_<module>.v*), a file *<module>_patched.v* is created. In addition, a *patchedDesign.v* file with the entire patched design is created. In patched designs, all design hierarchy and names are left unchanged, unless they are involved in the actual difference logic. The -ecoDir option may be used to specify an output path for *<module>_patched.v* files.

Note that there are important considerations for the ECO use model:

- The ECO design must be the A side and the design to be patched is the B side.
- For best results, the ECO design (A side) should be a technology mapped netlist.

GUI Access

Location:	Tools tab > **ECO Operations** > **Start ECO**
Action:	Initiate ECO Mechanisms.
Location:	Tools tab > **ECO Operations** > **Extract ECO**
Action:	Same as -eco generate using command line.
Location:	Tools tab > **ECO Operations** > **Finalize ECO**
Action:	Same as -eco final using command line.
Location:	Tools tab > **ECO Operations** > **Add Correspondence Constraint**
Action:	Add a correspondence constraint.
Location:	Debug Tab > **Difference Region Analysis**
Action:	Same as -eco generate using command line.

Examples

The user first runs the FormalPro tool to create a cache with logic differences.

```
formalpro -a changed_design.v  -b previous.v -common -slib library.lib
```

The user then performs difference region and ECO analysis:

```
$FORMALPRO_HOME/bin/formalpro -eco generate -cache <nonDefaultCacheDir>
```

If the user wishes to debug the differences, then invoke the formalpro -gui option:

```
$FORMALPRO_HOME/bin/formalpro -gui
```

If the user wishes to perform an ECO of a laid-out design:

For each *unmappedECO_<module>.v* in the *<cache>/eco* directory:

Use synthesis to create a corresponding *mappedECO_<module>.v* file. Create the patched module files (*<module>_patched.v*) and design (*patchedDesign.v*). Place these files into the *<cache>/eco* location, then run the -eco final command:

```
$FORMALPRO_HOME/bin/formalpro -eco final
```

Finally, depending on the layout tool ECO capabilities, use the patched designs files to accomplish the ECO.

Related Topics

-ecoDir

-tlist

extracteco

eco_correspond

-ecoDir

Scope: Global

Alias: None

The -ecoDir option is used in conjunction with the “-eco final” option. Use the -ecoDir option to specify a directory pathname where final patched modules and design netlist files are written.

Usage

- -ecoDir *directoryPathname* — Optional option specifying a directory where final patched design and modules are written.

Description

The -ecoDir option is used with the “-eco final” option to specify a directory where final patched design and modules netlists are written. By default, the files are written into the

<cache>/eco directory. If *directoryPathname* exists, the new files are written into it, perhaps overwriting files already in that directory. If *directoryPathname* does not exist, it is created.

GUI Access

The output directory path name can be applied interactively within the debugger tool using the following command:

```
extracteco -final directoryPathName
```

Examples

In this example, the final module and design patch files are written into an alternate directory, *outDir*.

```
$FORMALPRO_HOME/bin/formalpro -eco final -ecoDir ./ecoOut
```

Related Topics

-eco

-edifFile

Design-specific

Alias: edif

Specifies a gate-level EDIF file.

Usage

- -edif *filename* — Specifies a valid EDIF file to compile.

Description

This command compiles the specified file as EDIF, regardless of the file extension.

See Also

Suffix Control Switches (Library Files)	Switches that override the file extensions in *formalpro.ini* for a given type of library file.
Suffix Control Switches (Design Files)	Switches that override the file extensions in *formalpro.ini* for a given type of design file.

-sv	Overrides the Verilog file extension for design/library files in the given design scope (-a or -b), compiling the files as SystemVerilog files.
-sv2009	Overrides the file extension of the specified file, compiling the file as a 2009-formatted SystemVerilog file.
-vhdl2008File	Overrides the file extension of the specified file, compiling the file as a 2008-formatted VHDL file.
-verilogFile	Overrides the file extension of the specified file, compiling the file as a Verilog file.

GUI Access

Location:	**Project** tab > **A** tab
	Project tab > **B** tab
Action:	Select-**edif** from the dropdown menu to the left of the specific design file.

-encapsulateAll

Scope: Design-specific

Alias: None

Encapsulates all user-defined modules.

Usage

- -encapsulateAll — Enables encapsulation.
- -noencapsulateAll — Disables encapsulation (default).

Description

The **-encapsulateAll** command causes all user-defined modules to be encapsulated. It is logically equivalent to putting the following commands in the blackbox file:

```
encapsulate A *
encapsulate B *
```

Modules that are to be encapsulated have extra buffers added to the input and output ports for every instance of that module. Unlike other hierarchical approaches FormalPro solves every instance of a module separately.

Before encapsulation, FormalPro generates its normal flattened net list so as to allow constant values to propagate across module boundaries. The buffers that were added at the module boundaries are then used to create new comparison points and independent variables. This is

done using the same mechanism as the “make_pi” and “make_po” constraints. The new po's and pi's have names that are based on the instance name and the port name of the encapsulated module. These names have been found to be the most likely to match up between the A and B designs.

The new po's and pi's are then matched up using FormalPro’s usual matching system. There is one difference, however. If a make_po or make_pi cannot be matched, FormalPro will reconnect the two points. The reporting system tells you that the ports were not matched but that the unmatched ports will be benign.

There are many reasons why a module in one design would have ports that don't exist in the other. For example, the two designs might have different hierarchical structures such that what is done in a single module in one design is done in two modules in the other. Reconnecting the ports will automatically fold in the hierarchy boundary.

Another reason might be test circuitry. Consider an ATPG system that exists only on the B side. In a flattened comparison, the ATPG signal is forced to be disabled and the usual constant propagation will ensure that it doesn't cause differences. However, the enable signal has to reach every register and this signal may be added to every module interface in the design. It won't exist in the original design, so FormalPro won't be able to match it up. Once again, reconnecting it will prevent any problems in solving.

GUI Access

Location:	**Options** dialog box — **Blackbox** pane
Action:	Enable: **Select all modules** Disable: **Unselect all modules**

Examples

```
formalpro -encapsulateAll
```

-f

Global

Alias: None

Specifies an external file of command line switches used to invoke FormalPro.

Usage

- -f *command_file* — Specifies a command file.

Description

The **-f** switch specifies a text file of frequently used command line switches. For example:

```
// commented line
-cache ./my_formalpro.cache
```

In the command file, enclose environment variable names in curly brackets. For example,

```
${PWD}rest/of/the/path
```

If the boundary between the environment variable and a concatenated string is marked by a delimiter, the curly braces are optional. For example, the following definitions are equivalent:

```
${PWD}/rest/of/the/path
$PWD/rest/of/the/path
```

Environment variables are not expanded in the command line output in the log file.

Insert comments into the command file as follows:

- # and // — FormalPro treats everything after the comment character, up to a newline, as a comment.
- /* */ — FormalPro treats all characters between these comment characters as a comment.

GUI Access

Location:	**Options** dialog box — **General** pane > **Control** tab > **Other Options** entry box
Action:	Type the switch and argument as they would appear on the command line.

Examples

The following command_file:

```
// common_switch.txt -- commonly used switches
-cache ./my_fp.cache
```

translates to the following invoke commands:

```
formalpro -f common_switch.txt \
          -a design_a.v \
          -b design_b.v

formalpro -cache ./my_fp.cache \
          -a design_a.v -b design_b.v
```

-fastVerilogRead

Design-specific

Alias: None

Enables fast processing of Verilog netlist files for the A, B, or common design scopes.

Usage

- -fastVerilogRead — Optional switch that speeds up the verification of Verilog input files by only processing the sections that declare modules targeted for verification.
- -noFastVerilogRead — Disables *fastVerilogRead* mode and processes the entire contents of all the input Verilog files before extracting the needed module hierarchy. Default.

Description

The *fastVerilogRead* mode enables the FormalPro gate-level Verilog compiler to speed up the processing of large netlist files when a sub module is specified. This is very useful when there is a large Verilog netlist that only needs a small portion of the modules loaded for verification.

When enabled, FormalPro reads netlists in *fastVerilogRead* mode initially, but the following conditions can cause FormalPro to revert to full-file processing:

- All of the modules declared in all the input files are targeted for the verification run.
- No top-level module is specified.
- The -rtl compile mode is specified.
- Every module targeted for verification is declared at the top of its own file.
- The input file is encrypted or compressed, which will cause all of its contents to be read in.

GUI Access

None

Examples

The following example enables the *fastVerilogRead* mode for design A:

```
formalpro -a -mod uart_top design_a.v -fastVerilogRead \
          -b -mod uart_top design_b.v
```

-fl

Design-specific

Alias: None

Loads design files from the specified text file.

Usage

- -fl *designFileList* — Specifies the name and location of the test file.

Description

This switch allows you to specify which files are used in the verification through the use of a text file. This is useful for, but not specific to, designs using VHDL and mixed-language design flows (due to their order-dependence).

You can also specify your design files individually by using the designFile argument.

Within the designFileList, you can specify other file-specific switches, such as +incdir+, +define+, -work, -87, -93**,** -2008, -svFile, -vhdlFile, -verilogFile. The designFileList only accepts the # comment character, which treats everything between it and the end-of-line as a comment. You can specify environment variables within the designFileList.

GUI Access

Location:	**Project** tab > **A** tab **Project** tab > **B** tab **Project** tab > **Common** tab
Action:	1. Type the path to your file list or use:  2. Select **fl** from the dropdown menu to the left of file list path.

Examples

```
formalpro -a -fl design_a.fl \
          -b ./gate/design_b.v
```

where design_a.fl is:

```
# design file list for design A
-87 ./rtl/ser_out_mux.vhd
-work myworklib
-vhdlFile ../rtl/myVhdlFile.vhl
-87 ./rtl/status_registers.vhd
-93 ./rtl/transmit_rcv_control.vhd
-work work
-87 ./rtl/Amodule_top.vhd
-87 ./rtl/chaplin.vhd
./rtl/shadow.v
./rtl/artig.v
```

-flow

Scope:Global

Alias: None

Loads a built-in or custom flow file.

Usage

- -flow {customFile | builtinFlow}

  customFile — Name of the custom flow file. A path must be supplied if the file is not in a searched location. For more information, see “Creating a Custom Flow File” on page 61.

  builtinFlow — Built-in flow file. Options include synthesis vendors, certain features, and solve engine files. Vendor types overwrite each other and solve engine flows overwrite each other. The contents of a flow file can be read from the GUI by opening the flow named in the command line.

  - Oasys - Supports the Oasys-RTL tool
  - readVSDC - Supports in-line reading of guide files for the Oasys-RTL and Design Compiler tools. For more information, see “readVSDC Flow File” on page 297“.
  - Precision - Supports Precision RTL and RTLPlus
  - dc_ultra - Supports Synopsys Design Compiler
  - ibmlssd - Supports gate-level designs optimized with LSSD post-synthesis processing. FormalPro converts LSSD-style registers in your RTL design into latch pairs (master/slave configuration).
  - ti_asic - Provides instructions to the FormalPro Library Compiler specific to Texas Instruments ASIC libraries.
  - CadenceRC - For Cadence RTL/gate compare

- Nitro-SoC - For Mentor Olympus/Nitro router check gate/gate
- XST - For Xilinx XST or Vivado synthesis RTL/Gate compares
- xilinx_g2g - For Precision/Synplify netlist compared to Xilinx route
- altera_g2g - For Precision/Synplify netlist compared to Quartus route
- actel_g2g - For Precision/Synplify netlist compared to Actel/Microsemi route
- Synplify_Actel - For Synplify RTL/gate when Actel/Microsemi parts used
- DSP2 - Solve engine file. Good for digital filters.
- ExtSAT - Solve engine file. Uses an academic SAT solver to replace default
- retime - Use this in a Precision FPGA flow to resolve normal insertions of RAM and DSP macros in a Xilinx flow. For more information, see “-retime” on page 111.

  Note

  The Precision retime option is not enabled with the FormalPro retime support. The Precision “retime” and “gated clock” functions are also not supported by this option.

  For optimal Xilinx DSP performance, an added Precision option may be required. See current application notes.

Description

A flow file contains command line switches that alter the default settings for a particular run. Multiple *-flow* switches are processed in the order they are specified on the command line. For example, if you specify two flow files that contain settings for the same switch, the last flow file read is the one that is used. Settings explicitly entered on the command line override flow-file settings.

When multiple -flow files are specified, they are cumulative. For example:

```
formalpro -flow oasys -flow retime
```

The built-in flow files are located in the following directory:

```
$FORMALPRO_HOME/pkgs/fv/userware/default/flows
```

Creating a Custom Flow File

Use the following steps to create a custom flow file:

1. Save a copy of the $FORMALPRO_HOME/lib/formalpro.ini file to your home directory.

2. Delete the settings not needed for your flow file. Do not delete any settings from the [Flow], [A], and [B] sections.
3. Modify the remaining lines to set the options for your custom flow. Be sure to maintain the sections and format of the .ini file. For more information, see Initialization File in the *FormalPro User's Manual*. Refer to the default formalpro.ini file for examples of option settings.
4. Save the file to a unique name with the .ini suffix in one of the locations searched by FormalPro. FormalPro automatically searches the following locations in the order specified:
   a. current working directory
   b. your home directory
   c. directory specified with the $FORMALPRO_FLOW environment variable

Note

Do not save the file within a formalpro.cache or in the $FORMALPRO_HOME tree.

GUI Access

Location: **Options** dialog box —
General tab > **Flow selection** dropdown box.

Action:
1. Select a flow file.
2. Click **Add** button. Once added, right-click on the flow file to read it.

Examples

The following example uses a custom flow file to specify VHDL work libraries.

The custom flow file mylibs.ini specifies the libraries:

```
[library]
m3s001lib = $HOME/demos/demo_2.3/m3s001lib
m3s003bolib = $HOME/demos/demo_2.3/m3s003bolib
```

The custom flow file is loaded as follows:

```
formalpro -flow mylibs.ini -a a.vhd -b b.vhd
```

-formalEyes

Scope:Design-specific

Alias: None

Reports potential design problems.

Usage

- -formalEyesFloat— checks for undriven or floating nets
- -formalEyesMulti— checks for multi-driven net contention
- -formalEyesX— checks for capture of X assignments
- -formalEyesConstRegs— checks for constant registers
- -formalEyesAll— enables all FormalEyes checks

Description

The FormalEyes commands detect and report design conditions that have the potential to cause unintended circuit behavior. Many of the potential design errors that FormalEyes looks for are difficult to detect with traditional verification techniques such as simulation of functional vectors. FormalEyes uses formal, structural, and functional analysis techniques to identify objects that are replicated, redundant, optimizable, or constant.

FormalEyes can identify and report the following types of potential design errors:

- non-toggle registers
- no-path registers
- constant registers
- constant state vector bits
- multiply driven net contention
- X assignment captured
- undriven/floating net captured
- redundant mux logic assignments

Typically, when an error manifests itself, it causes one or more of these items to occur. For example, an error in bus-control logic might present itself as a bus contention error or as a floating bus error. Another example would be an error in the state-transition logic of a FSM that presents itself as a constant state vector bit or as a non-toggle register. In RTL code, an X assignment that was never intended to be captured in a register becomes capturable.

Using FormalEyes to find and report potential misbehavior when an equivalency check difference occurs gives you a convenient way to screen the reported events to determine if an unintentional event has occurred, and enables you to concentrate on fixing any design errors.

While FormalEyes generates reports that identify *potential* misbehavior, it is up to you to decide whether or not the event is actually an *intended* behavior. For example, a designer could have intended for particular state vector bits to be constant, and FormalEyes would still report the event as a potential misbehavior.

While FormalEyes is capable of uncovering some types of design misbehaviors, the designs should not be considered free of these defects when no defects are reported by FormalEyes. FormalEyes puts forth a “good” effort to uncover these problems, but some of these errors may require more time to uncover than what is currently allocated to FormalEyes within the Formalpro tool.

GUI Access

Location: **Options** dialog box —
A specific pane > **FormalEyes** tab >
Bspecific pane > **FormalEyes** tab >

Action: Enable: Click the **Set All** button or select specific checks
Disable: Click the **Clear All** button

Examples

The following example will check design A only for captured floating nets and captured X assignments, whereas design B will have all checks run on it:

```
formalpro -a -formalEyesFloat -formalEyesX -b -formalEyesAll
```

-fpga

Scope:Global

Alias: None

Specifies the FPGA vendor.

Usage

- -fgpa {altera | actel | xilinx}

Description

This switch specifies the FPGA vendor. When invoking FormalPro using formalpro_fpga, the vendor must be specified, and the corresponding vendor-qualified license must be available, in order to start a verification run.

GUI Access

Location:	**Project** tab > **Options** button > **Compile** tab > **FPGA technology** field
Action:	Use the drop-down to select **Altera**, **Actel,** or **Xilinx**

Examples

```
$FORMALPRO_HOME/bin/formalpro -fpga xilinx \
   -a \
   -y $FORMALPRO_HOME/pkgs/fv/lib/replacementLib/xilinx \
   -y <path>/unisims -mod <top_module> -rtl <synth_netlist>\
   -b \
   -y $FORMALPRO_HOME/pkgs/fv/lib/replacementLib/xilinx \
   -y <path>/simprims \
   -rtl <place_route_netlist> \
   -mod <top_module>
```

-FSMencoding

Scope:Design-specific

Alias: None

Specifies that Finite State Machines in an RTL design should be encoded with the specified scheme during the Compile stage.

Usage

- -fsmencoding {ignore | auto | binary | gray | onehot | random}
  - ignore — compiles using binary encoding (default).

    The only difference is when your Verilog design contains a parameter statement containing a //synopsys enum directive, for example:

    ```
    parameter [2:0] //synopsys enum days Mon=3'b010, Tue =3'b110,
    Wed=3'b001;
    reg [2:0] /* synopsys enum days */ WeekDays;
    ```

    FormalPro encodes the states as specified in the parameter statement.
  - auto — behaves similar to the 'onehot' argument.
  - binary — compile using binary encoding.
  - gray — compile using gray-code encoding.
  - onehot — compile using one hot encoding.
  - random — compile using random binary code assignment.

Description

When you specify a FSM encoding scheme for your RTL design, FormalPro applies the scheme to all FSMs found in both the A and B designs. You will typically use this switch in an RTL-to-Gate run.

GUI Access

Location: **Options** dialog box —
A specific pane > **RTL** tab >
B specific pane > **RTL** tab >
FSM encoding dropdown box

Action: Select argument

Examples

```
formalpro -a -fsmencoding onehot./rtl/design_a.vhdl -b ./gate/design_b.v
```

-gate

Scope: Design-specific

Alias: None

Specifies that a design scope consists completely of gate-level design files.

Usage

- -gate

Description

FormalPro, by default, parses your design files to determine if they are RTL- or gate-level. This switch explicitly informs FormalPro that a design is gate-level.

You should not specify this switch if a design is not completely gate-level format because FormalPro will black box any module containing RTL constructs.

GUI Access

Location: **Options** dialog box —
A specific pane >**Control** tab >
B specific pane > **Control** tab >
Design level dropdown box

Action: Select gate

Examples

```
formalpro -a ./rtl/design_a.vhd \
          -b -gate ./gate/design_b.v
```

-gatedClocks

Scope: Global

Alias: -gatedClock

Considers gated-clock structures during verification.

Usage

- -gatedClocks

Description

You should use this switch when you are verifying a *reference* design against a *modified* design on which you have performed power optimization. Typically, power optimization introduces gated-clock structures into the design using latches, which results in your modified design containing matchable objects that do not exist in your reference design.

In the above scenario, if you do not specify this switch, FormalPro classifies these latches in your *modified* design as unmatched objects, and declares targets in their fan-out as “fed by unmatched”.

When you specify this switch, the tool automatically determines if the latches result from gated-clock structures and can safely declare the latches as transparent. These latches are reported in detail in the report available from the **Reports > Removed > Simplified Registers** menu item.

GUI Access

Location: **Options** dialog box —
General pane > **Solve** tab

Action: Enable: Select **Gated clock designs**
Disable: Unselect**Gated clock designs**

Examples

```
formalpro -gatedclock
          -a ./rtl/design_a.vhd \
          -b ./gate/power_optimized_design_b.v
```

FormalPro reports any latches it declares as transparent, resulting from the use of this switch, in the file *formalpro.cache/reports/simplifiedRegs.report*, as shown:

```
...

5. Latches Used in Gated-Clock Structure( converted to transparent)
-------------------------------------------
\B.top.logic.fz80.reg.clk_gate_reg.latch.q_ppi
\B.top.logic.herc.m_blk.clk_gate_3_msb_reg.latch.q_ppi
\B.top.logic.fz80.arg.clk_gate_reg.latch.q_ppi
```

-noGateOptimization

Design-specific

Alias: None

Controls gate optimization.

Usage

- -gateOptimization — Optional switch that enables gate optimization. Can also be specified in the *formalpro.ini*. Default setting.
- -noGateOptimization — Optional switch that disables gate optimization. Can also be specified in the *formalpro.ini*.

Description

By default, FormalPro optimizes (converts and eliminates) the following gates:

- Gates with constant inputs. For example, a 2-input AND gate with one input tied high is translated to a BUF buffer.
- Mux-like gate logic containing a feedback path. When appropriate, this gate logic is replaced with a registered latch.

Examples

Example 1

The following example disables gate optimization:

```
formalpro -nogateoptimization
```

Example 2

The following example enables gate optimization:

```
formalpro -gateoptimization
```

-generics

Scope: Design-specific

Alias: -generic

Specifies the name and value of generics specified in the top-level module of your VHDL RTL design.

Usage

- -generics name=value
  - name — the name of a generic. FormalPro automatically converts this argument to all lowercase characters.
  - value — the value assigned to a generic name.

GUI Access

Location:	**Options** dialog box —
	A specific pane > **Generics** tab
	B specific pane > **Generics** tab
Action:	Enter “name=value” information. Separate entries by a space if you specify more than one generic.

Description

You can specify any number of generics on the command line in the following format.

```
-generics foo=1 -generics bar=2
```

Examples

```
formalpro -a -generics foo=10 \
            ./rtl/design_a.vhdl \
          -b ./gate/design_b.v
```

-gui

Scope: Global

Alias: None

Runs FormalPro in the GUI mode.

Usage

- -gui

Description

Use this switch to run FormalPro through the GUI.

You can specify global switches in addition to the -gui switch to pre-set the GUI with the global options.

FormalPro loads all information from the formalpro.cache, if one is present, into the GUI, allowing you to interactively analyze the previous run.

Examples

This example loads any existing formalpro.cache into the GUI:

```
formalpro -gui
```

This example starts the FormalPro GUI in FPGA-only mode, with Xilinx specified as the FPGA vendor:

```
formalpro_fpga -gui -fpga xilinx
```

This example loads the cache my.cache, if it exists, into the GUI:

```
formalpro -gui -cache my.cache
```

This example does not load any existing cache into the GUI, but does preset the GUI with the specified options:

```
formalpro -gui -loglevel full -cache my.cache
```

-help

Scope: Stand-alone

Alias: None

Displays text files with quick-reference help on several topics.

Usage

- -help [blackbox | match | constraints | rules] — when you specify this switch with no argument, you receive a help file for the formalpro command.
  - blackbox — displays help on creating black boxes.
  - match — displays help on creating explicit matches.

- constraints — displays help on writing constraint files.
- rules — displays help on writing implicit match rules.

Description

The help files displayed give a quick overview of the topics selected. You can also access these same files from within the GUI by selecting the Help menu.

GUI Access

Location: **Help** menu

Examples

```
formalpro -help match
```

-noheuristicNameLookup

Scope: Global

Alias: None

Disables the resolution of port/register names using an internal name database.

Usage

- -heuristicNameLookup — Automatically resolves names (default).
- -noheuristicNameLookup — Disables the automatic resolution of port/register names and returns errors when names do not resolve.

Description

The heuristicNameLookup feature uses a database generated at compile time to try and match unresolved names within constraint and match files. Only tests where name warnings currently occur should be impacted. This option is enabled for all default flows; If you find that it causes errors, you can disable it and manually correct the names involved.

All corrections are logged in the report file and a warning is output to the main log.

GUI Access

Location: **Options** dialog box —
General pane > **Match** tab

Action:	Enable: Select **heuristicNameLookup**
	Disable: Unselect **heuristicNameLookup**

Examples

```
formalpro -heuristicNameLookup -constraintFile ./vsdc.constraint \
-a design_a.v \
-b design_b.v
```

-ignoreNoPath

Scope: Global

Alias: None

Controls how FormalPro matches comparison points that do not have a path to a primary output.

Usage

- -ignoreNoPath — ignores these comparison points (default).
- -noignoreNoPath — attempts to match these comparison points.

Description

FormalPro reports these comparison points in the *ignorable_objects.report* file. You can access the report from the GUI by selecting the **Reports > Constraints > Ignorable objects** on the main menu.

GUI Access

Location:	**Options** dialog box —
	General pane > **Match** tab
Action:	Enable: Select **Ignore no path targets**
	Disable: Unselect **Ignore no path targets**

Examples

```
formalpro -noignorenopath \
          -a design_a.v \
          -b design_b.v
```

-inferVHDLorder

Scope: Design-specific

Alias: None

Disables the automatic determination of the order of VHDL design files in a design scope.

Usage

- -inferVHDLorder — activates this functionality (default).
- -noinferVHDLorder — deactivates this functionality.

Description

Deactivate this functionality to manually specify the order of your RTL design files.

GUI Access

Location:	**Options** dialog box —
	A specific pane > **RTL** tab
	B specific pane > **RTL** tab
Action:	Enable: Select **Infer VHDL order**
	Disable: Unselect **Infer VHDL order**

Examples

The following example shows how to instruct FormalPro to compile a directory of VHDL files and automatically infer the correct VHDL order of those files.

```
formalpro -a -infervhdlorder \
            -fl ./rtl/vhdl/filelist.fl \
          -b design_b.v
```

+incdir

Scope: Design-specific

Alias: None

Specifies the location of a directory referenced by a Verilog include statement.

Usage

- +incdir+include_dir

  include_dir — specifies the directory that contains your included files.

Description

You should use this switch if your Verilog RTL files contain ‘include statements and you want FormalPro to compile the code within the referenced file.

Your ‘include statements generally reference a file, but for this switch, you need to specify the directory that contains the file, not the file itself.

When you have multiple ‘include statements that point to different directories, you must specify +incdir+ for each directory.

The order in which you specify +incdir+ on the command line, or within file lists, determines the precedence order for FormalPro. In cases where ‘include files with the same name exist in more than one directory, FormalPro uses the first one encountered, as specified on the command line.

GUI Access

Location:	**Project** tab > **A** tab **Project** tab > **B** tab
Action:	Type the switch and argument on a new line, as you would a design file location.

Examples

```
formalpro -a +incdir+./rtl/inc_files/ \
            ./rtl/design_a.v \
          -b ./gate/design_b.v
```

-libConfigFile

Scope: Design-specific

Alias: -lcf

Specifies rules for replacing Verilog library cells.

Usage

- -libConfigFile *configFileName*

  configFileName — specifies the location of the text file containing the replacement rules.

Description

In some cases, Verilog technology libraries may contain behavioral or timing-check constructs that you need to remodel for FormalPro to compile them. Once you have obtained these remodeled libraries, you need to map the original libraries to the new libraries using a library configuration file.

The format of the library configuration file (*configFileName*) consists of one entry per line, where FormalPro ignores blank lines and comment lines prefixed by the number sign (#). Each entry line has the following Usage:

```
<cellToReplace> : <replacementCell> (<verilogPortList>);
```

where cellToReplace and replacementCell are cell names, separated by a colon (:), and verliogPortList is a comma-separated list of port names enclosed in parentheses. You must end the entry line with a semicolon (;), as shown in the following example:

```
# commented line
cellA : cellA_new (q, a, b);
```

You must load both the original library and the remodeled library with the library specification switches (-alib, -v, -y, or -slib).

GUI Access

Location: **Options** dialog box —

A specific pane > **Control** tab > **Other Options** entry box

B specific pane > **Control** tab > **Other Options** entry box

Action: Type the switch and argument as they would appear on the command line.

Examples

In the following example, design B relies on technology libraries that needed to be remodeled. The directory lib_orig contains all the original libraries and the directory lib_remodel contains the remodeled libraries (you must specify both directories in the command line).

```
formalpro -a ./rtl/design_a.v \
          -b -y ./lib_orig/ \
             -y ./lib_remodel/ \
             -lcf ./lcf.txt \
             ./gate/design_b.v
```

The file lcf.txt contains the replacement rules:

```
lib1 : lib1_mod (a, b, c, d);
lib2 : lib2_mod (a, b, c, d);
```

-LibertyPGpins

Either include or exclude power and ground pins declared on a CELL in a Liberty library.

Usage

- -LibertyPGpins — Includes power and ground pins.
- -noLibertyPGpins — Excludes power and ground pins.

Related Topics

fplibcomp

+libext

Scope: Design-specific

Alias: None

Specifies which extensions the tool should accept for library files, and their order of precedence.

Usage

- +libext+extension[+extension...]

  extension — a file extension, with or without the period (.) character.

Description

You should use this switch along with -y for controlling library resolution.

The order in which you specify extension(s) determines the precedence when the tool encounters two library cells of the same name.

Refer to the section "Verilog Library Resolution" in the FormalPro User's Manual for further information.

GUI Access

Location: **Options** dialog box —
A specific pane > **Control** tab >
B specific pane > **Control** tab >
Verilog library extensions entry box

Action: Type in the list of extinctions, in order of precedence.

Examples

```
formalpro -a designa.v -b designb.v \
            -common -y ./verilog_lib/ +libext+.v+.V+.vg
```

+liborder

Scope: Design-specific

Alias: None

Specifies the order in which the tool searches libraries for unresolved modules.

Usage

- +liborder

Description

If the unresolved module is located in a design file, the tool initiates the search in the library file (-v) or directory (-y) that immediately follows the design file on the command line.

If the unresolved module is located in a library, the tool initiates the search within that library.

If the module remains unresolved after the initial search, the tool then searches the next library listed on the command line.

You cannot specify this switch along with +librescan.

Refer to the section "Verilog Library Resolution" in the *FormalPro User's Manual* for further information.

GUI Access

Location:	**Options** dialog box —
	A specific pane > **Control** tab
	B specific pane > **Control** tab
Action:	Enable: Select **+liborder** button
	Disable: Select desired option button

Examples

```
formalpro -a ./rtl/designA.v -b +liborder \
               ./gate/designb_1.v -v ./lib/lib1.v \
               ./gate/designb_2.v -v ./lib/lib2.v
```

+librescan

Design-specific

Alias: None

Specifies the order in which the tool searches libraries for unresolved modules.

Usage

- +librescan

Description

If the unresolved module is from a design file or a library file, the first library in the design scope is searched. If after this initial search the module remains unresolved, the next library listed in the design scope is searched.

If the unresolved module is from a library directory, the library directory is searched. If after this initial search the module remains unresolved, the first library listed in the design scope is searched.

You cannot specify this switch along with +liborder.

For more information, see Verilog Library Resolution in the *FormalPro User's Manual*.

GUI Access

Location:	**Options** dialog box —
	A specific pane > **Control** tab
	B specific pane > **Control** tab
Action:	Enable: Select **+librescan** button
	Disable: Select desired option button

Examples

```
formalpro -a ./rtl/designA.v -b +librescan \
              ./gate/designb_1.v -v ./lib/lib1.v \
              ./gate/designb_2.v -v ./lib/lib2.v
```

+libVerbose

Design-specific

Alias: None

Creates additional error and warning information for Verilog libraries.

Usage

- +libVerbose

Description

The tool outputs additional information in the logs available from the **Logs > Compile > Compile Details A|B** menu items.

Refer to the section "Verilog Library Resolution" in the *FormalPro User's Manual* for further information.

GUI Access

Location: **Options** dialog box —

A specific pane > **Control** tab

B specific pane > **Control** tab

Action: Enter switch in **Other design A|B options** text entry box

Examples

```
formalpro -a ./rtl/designA.v -b +librescan +libverbose \
              ./gate/designb_1.v -v ./lib/lib1.v \
              ./gate/designb_2.v -v ./lib/lib2.v
```

-log

Global

Alias: None

Renames the FormalPro log files.

Usage

- -log *logFileName*

  logFileName — String that specifies a log file name.

Description

This switch renames the file formalpro.log to <logFileName>.log, as well as the log files for the different log levels (.mini, .compact, and .full).

GUI Access

Location:	**Project** tab > **General** tab > **Log file** entry box
Action:	Type the new name of the log file or use: 

Examples

```
formalpro -log new_fp \
            -a design_a.v \
            -b design_b.v
```

Produces the file new*_fp.log* in the current directory, and the files *new_fp.mini*, *new_fp.compact*, and *new_fp.full* in the logs subdirectory of *formalpro.cache*.

-logLevel

Global

Alias: -ll

Controls the amount of information written to the FormalPro log file and stdout.

Usage

- -logLevel [mini | compact | full]

  mini —writes only the final comparison status.

  compact —writes summary information for each stage. Default.

  full — writes all detailed information for each stage.

Description

FormalPro always generates all three log files for each run. You can find them at the following FormalPro cache location:

```
formalpro.cache/logs/
   formalpro_mini.log
   formalpro_compact.log
   formalpro_full.log
```

You can alter the filenames with the -log switch. You can also access these files from the **Logs** drop-down menu in the GUI.

GUI Access

Location: **Options** dialog box —
General pane > **Control** tab > **Log level** dropdown box.

Action: Select argument.

Examples

The following example specifies that the FormalPro log should contain all detailed information for the run.

```
formalpro -loglevel full -a design_a.v -b design_b.v
```

-masterSlaveMerge

Global

Alias: None

Compiles library files containing two latches in a master/slave configuration so that the register is represented as a single DFF.

Usage

- -masterSlaveMerge — enables this functionality (default).
- -nomasterSlaveMerge — disables this functionality.

Description

This switch allows you to compile your RTL design in the same way you synthesized your design.

GUI Access

Location: **Options** dialog box —
General pane >**Lib** tab

Action: Enable: Select **Merge Master slave**
Disable: Unselect **Merge Master slave**

Examples

```
formalpro -nomasterslavemerge \
           -a ./rtl/design_a.v -b ./gate/design_b.v
```

-matchFile

Global

Alias: None

Specifies the location of a match file containing explicit match commands.

Usage

- -matchFile *filename* — specifies the location of the match file.

filename — a file location. Non-literal pathnames are relative to the current directory.

- -nomatchFile — instructs FormalPro to ignore a specified match file when restarting a previous run.

Description

For further information about explicit matching, refer to the section “Match Files” on page 210.

GUI Access

Location:	**Project** tab > **General** tab > **User match** entry box
Action:	Type the path to your user match file or use: 

Examples

```
formalpro -matchfile ./setup/match.cmd \
          -a design_a.v \
          -b design_b.v
```

-matchseq

Global

Alias: -match

Specifies the algorithms used during the match stage.

Usage

- -matchseq algorithm[:algorithm]...

  Optional switch that specifies a colon separated list of the matching algorithms applied during a run. The algorithms are applied in the order listed. By default, the *user:name_da:func* algorithms are used.

You must specify all applicable arguments. For example, if you only specify *user*, FormalPro does not perform *name* matching using the default rule file. Algorithm options include:

- **user** — *explicit* memory element and port matching based on a custom match file specified with the -matchFile switch. If you specify an explicit match file with -matchFile, but do not specify this argument, FormalPro automatically runs this algorithm before any other algorithms.
- **name** — *implicit* name matching based on a custom rule file specified with the --ruleFile switch. If no custom rule file is specified, the default rule file is used.
- **name_na** — name matching on registers similar to the *name* argument except only canonical names are matched, not register aliases.
- **name_da** — same as the *name* argument except a first run matches canonical names, and a second run matches alias names from any unmatched names.
- **name_oa** — name matching on registers similar to the *name* argument except only register alias names are matched.
- **graph** — matching based on graph isomorphism.
- **func** — matching on registers based on functional techniques. This argument should always follow the *name* argument in the algorithm sequence.
- **func_mp** — matching on registers based on multi-pass functional techniques. This argument should always follow the *name* argument in the algorithm sequence. This algorithm may cause excessive runtime on a large design with many unmatched registers and should only be used if the *func* algorithm fails to achieve the desired results.

Description

FormalPro matches elements by applying several matching algorithms. You can use this option to control which matching algorithms are used for a run and their order. For more information, see the section "Completing Matching" in the *FormalPro User's Manual*.

GUI Access

Location: **Options** dialog box —

General pane > **Match** tab > **Match sequence** pane.

Action:

1. Select from the **Match sequence** dropdown box the first algorithm FormalPro should run.
2. Click the **Add** button
3. Repeat steps 1 and 2 for each successive algorithm.

Examples

The following example instructs FormalPro to perform explicit matching (*user*) with the rules from the file *match.user* and implicit rule matching on canonical names only (*name_na*).

```
formalpro -matchseq user:name_na -matchfile match.user \
            -a design_a.v \
            -b design_b.v
```

-memLimit

Global

Alias: None

Specifies the memory limit FormalPro should use when executing Binary Decision Diagrams (BDD).

Usage

- -memLimit *number*

  number — the memory limit in units of megabytes (MB).

Description

In some cases, HP-UX systems may report the incorrect amount of memory to FormalPro; when this occurs, you can manually set the amount of memory that FormalPro uses.

GUI Access

Location: **Options** dialog box —
General pane > **Solve** tab > **BDD memory limit** dropdown box.

Action: Select argument.

Examples

```
formalpro -memLimit 512 \
            -a design_a.v \
            -b design_b.v
```

-mergeReplicatedReg

Global

Alias: None

Facilitates matching and solving of replicated registers.

Usage

- -mergeReplicatedReg

Description

Instructs FormalPro to look for instances of replicated registers so they can be matched with the appropriate single register on the opposite design side.

An example where this option is useful is when a single register in an RTL-level design is replicated in the gate-level design to facilitate a layout that meets timing requirements. With the -mergeReplicatedReg option, FormalPro is able to match the two registers in the gate-level design with the single register in the RTL design.

GUI Access

Location:	**Options** dialog box — **General** pane > **Options** > **Match** tab
Action:	Enable: Select **Merge replicated registers** Disable: Unselect **Merge replicated registers**

Examples

```
formalpro -mergeReplicatedReg -a design_a.v \
          -b design_b.v
```

-mod

Design-specific

Alias: -top

Specifies the top-level design module.

Usage

- -mod {*moduleName* | *libName*.*entName*(*archName*) | -*entName*(*archName*) *entName*}
  - *moduleName* — Name of the top-level module for a Verilog or EDIF design.
  - *libName* — Name of the VHDL library containing the entity. The default value is work.
  - *entName* — Name of the VHDL entity.
  - *archName* — Name of the architecture the VHDL entity uses. By default, the most recently read architecture is used.

Description

In some instances, where there are multiple top-level modules or entities, you need to specify which module FormalPro should recognize as the top-level. This switch also allows you to perform a verification on sub-modules of the hierarchy.

GUI Access

Location: **Project** tab >—
A tab > **Top level** entry box
B tab > **Top level** entry box
Common tab > **Top level** entry box

Action: Enter module or entity name using command line syntax.

Examples

```
formalpro -a -mod uart_top design_a.v \
          -b -mod uart_top design_b.v

formalpro -a -mod "pkg.uart_top(synth)" design_a.vhd \
          -b -mod uart_top design_b.v
```

-mp

Global

Alias: None

Enables and configures verification processing across multiple CPUs.

Usage

- -mp*integer*
  - *integer* — Required integer that enables multi-processor (MP) mode and specifies the number of CPUs to use for the verification run.

Description

The MP mode divides up the processing associated with the compilation and solve phases of a verification run into jobs and then assigns the jobs to different CPUs. You can only use the MP mode for multiple CPUs within a single machine environment.

GUI Access

Location: **Options** dialog box —
General pane > **Control** tab >
Number of processes dropdown box.

Action: A) Select argument or
B) Type number

Examples

Both of the following commands invoke FormalPro and enable MP mode with 4 CPUs:

```
formalpro -mp 4 -a design_a.v -b design_b.v
```

```
formalpro_fpga -mp 4 -a design_a.v -b design_b.v
```

-mpLimit

Global

Alias: None

Specifies the max number of jobs for CPUs in multi-processing (MP) mode.

Usage

- -mpLimit *integer*

integer — Specifies the maximum number of jobs that can run on each CPU. The default value for -mpLimit is 16 jobs. Set to 1 for the old MP behavior.

Description

The MP mode divides up the processing associated with the compilation and solve phases of a verification run into jobs and then assigns the jobs to different CPUs. You can only use the MP mode for multiple CPUs within a single machine environment.

GUI Access

Location: Options dialog box —
General pane > Control tab >
Number of processes dropdown box.

Action: A) Select argument or
B) Type number

Examples

The following command invokes FormalPro, enables MP mode with 4 CPUs and allows up to 10 jobs per CPU with a minimum time limit of 200 seconds for processing each job:

```
formalpro -mp 4 -mpLimit 10 -mpTimeLimit 200 -a design_a.v -b design_b.v
```

-mpTimeLimit

Global

Alias: None

Limits amount of processing time for each CPU in multi-processing (MP) mode.

Usage

- -mpTimeLimit *integer*

integer — Specifies the minimum amount of time, in seconds, allowed for each CPU to process a job. The default value is 300 seconds.

Description

The MP mode divides up the processing associated with the compilation and solve phases of a verification run into jobs and then assigns the jobs to different CPUs. You can only use the MP mode for multiple CPUs within a single machine environment.

GUI Access

Location:	**Options** dialog box —
	General pane > **Control** tab >
	Number of processes dropdown box.
Action:	A) Select argument or
	B) Type number

Examples

The following command invokes FormalPro, enables MP mode with 4 CPUs and allows up to 10 jobs per CPU with a minimum time limit of 200 seconds for processing each job:

```
formalpro -mp 4 -mpLimit 10 -mpTimeLimit 200 -a design_a.v -b design_b.v
```

-multiplierArchitecture

Design-specific

Alias: None

Specifies the architecture used for compiling all multipliers in RTL designs.

Usage

- -multiplierArchitecture architecture[_adderType][_swap]

  The default setting is based on the design and flow type.

  architecture — specifies the multiplier architecture. Options include:

  - wall — Wallace tree architecture
  - csa — Carry-save Adder architecture. Default for FPGAs.
  - str — pipelined Wallace tree architecture
  - nbw — non-Booth Wallace architecture. Default for Oasys-RTL flows.
  - mcarch — architecture of Synopsys Module Compiler. There is no associated adder type.
  - csmult — Synopsys DesignWare multiplier.
  - pparch — Synopsys DesignWare multiplier. Default for ASICs.
  - adderType — specifies the final adder architecture within the multiplier. Options include:
  - cla — Carry Look Ahead adder (default for csa)
  - rpl — Ripple adder
  - csel — Carry Select adder
  - csm — Conditional Sum Module adder (default for wall)
  - bk — Brent Kung adder (default for nbw)
  - swap — specifies the swap order of the operands. Options include:
  - axb — compile the operands as specified in the RTL (default)
  - bxa — swap the order of the operands before compiling

  Note

  Each time you change the multiplier architecture with this option, you must recompile the design.

Description

This switch specifies that all multipliers in an RTL design are of a specific architecture. This information instructs the compile stage on how to generate the internal database to most closely resemble your gate-level design.

This switch is overridden, by default, if a multiplier is declared as a specific architecture through the use of pragmas (Verilog) or attributes (VHDL) within the RTL.

For more information see the following topics:

- Specifying Multiplier Architectures in the FormalPro User's Manual
- multiplierarchitecture

GUI Access

Location: **Options** dialog box —
A specific pane > RTL tab >
B specific pane > RTL tab >
Multiplier architecture dropdown menu, and
Multiplier final adder dropdown menu, and
Multiplier operands swap dropdown menu

Action: Select arguments.

Examples

For this example, FormalPro will compile all multipliers in your RTL design with the non-Booth-Wallace architecture, where the final adder in the multiplier is a Ripple adder.

```
formalpro -a ./rtl/design_a.v -multiplierarchitecture nbw_rpl \
          -b ./gate/design_b.v
```

Other examples are:

```
-multiplierarchitecture wall_csel_bxa
-multiplierarchitecture csa_rpl_axb
-multiplierarchitecture csa_bxa
```

+noLibCell

Design-specific

Alias: None

Treats Verilog files specified with -v or -y as design cells.

Usage

- +nolibcell

Description

When an RTL design is a part of your verification run, this switch allows you to perform debugging and cross-probing when you specify any design files with the -v or -y library resolution switches.

Refer to the section "Verilog Library Resolution" in the FormalPro User's Manual for further information.

GUI Access

Location:	**Options** dialog box —
	A specific pane > **Control** tab
	B specific pane > **Control** tab
Action:	Enter switch in **Other design A|B options** text entry box

Examples

```
formalpro -a ./rtl/designA.v \
          -b +nolibcell \
               ./gate/designb_1.v -v ./lib/lib1.v \
               ./gate/designb_2.v -v ./lib/lib2.v
```

-optimizeEqOpers

Design-specific

Alias: None

Prunes redundant bits from large equivalence operators during compilation.

Usage

- -optimizeEqOpers — enables this functionality.
- -nooptimizeEqOpers — disables this functionality. Default.

GUI Access

Location: **Options** dialog box —
A specific pane > **RTL** tab >
B specific pane > **RTL** tab >
Optimize equivalent operator check box

Action: Use check box to disable/enable.

Examples

The following example enables the *optimizeEqOpers* functionality:

```
formalpro -optimizeEqOpers \
```

-noOverWrite

Global

Alias: None

Determines whether any pre-existing FormalPro cache is overwritten at the start of the run.

Usage

- -overWrite (default)
- -noOverWrite

Description

At the start of each run, FormalPro deletes any existing cache and creates a new one. Also removed is any existing *formalpro.log* file.

Use **-noOverWrite** to preserve a pre-existing *formalpro.cache*. If the **-noOverWrite** option is used, and the *formalpro.cache* is present, FormalPro exits and displays a warning. If no *formalpro.cache* exists, a new one is created.

The -restart option overrides **-overWrite**, leaving the existing cache. In this case, any new output files created during the run overwrite the corresponding files in the existing cache.

Use the -cache switch to preserve a pre-existing cache or use the -archive switch to save the cache in a compressed form.

GUI Access

Location: **Options** dialog box —
General pane > **Control** tab

Action: Enable: **Select Overwrite**.
Disable: **Unselect Overwrite**.

Examples

```
formalpro -nooverwrite \
            -a design_a.v \
            -b design_b.v
```

-parameters

Design-specific

Alias: -parameter

Specifies the name and value of parameters specified in the top-level module of your Verilog RTL design.

Usage

- -parameters name=value
  - *name* — Name of a parameter. This argument is case-sensitive.
  - *value* — Value assigned to a parameter name.

Description

You can specify any number of generics on the command line in the following format.

```
-parameters foo=1 -parameters bar=2
```

For Verilog designs, this switch passes parameters by name rather than by position, so you cannot specify -parameters “(1,2)”.

GUI Access

Location: **Options** dialog box —
A specific pane > **Generics** tab
B specific pane > **Generics** tab

Action: Enter “name=value” information. Separate entries by a space if you specify more than one parameter.

Examples

```
formalpro -a -parameters foo=10 \
              ./rtl/design_a.v \
          -b ./gate/design_b.v
```

-PACheck

Global

Alias: None

Enables the Power Aware (PA) checker and controls the types of power-aware cells included in the report.

In previous releases, this option was named pmCellCheck.

Usage

- -PACheck {**none** | all | [isolation] : [level_shifter] : [retention]}
  - none — no report generated (default).
  - all — reports all power-aware mismatches.
  - More than one of the following may be specified, in any order, separated by a colon (:)
    - isolation — reports isolation cells mismatches.
    - level_shifter — report level_shifter mismatches.
    - retention — reports only mismatches between retention cells.

GUI Access

Location: **Options** dialog box —
General pane > **Match** tab

Action: Select which type of check to run.

Examples

```
formalpro -PACheck all
```

```
formalpro -PACheck isolation:level_shifter
```

-paConfigFile

Design-specific

Alias: None

Assigns power-aware (PA) type attributes to library cells.

In previous releases, this option was named pmConfigFile.

Usage

- -paConfigFile configfile

  configfile— the name (or path and name) of a file containing cell-to-type mappings. Non-literal pathnames are relative to the current directory.

Description

Use this option to assign a PA type attribute (a string) to library cells. During the FormalPro compilation, the attribute is assigned to each instance of the library cell. The attribute is used by the Formalpro PA checker to identify and match PA objects.

The file *configfile* contains one or more statements in the following format:

cell_type **LIBCELL,** cell_name

where *cell_name* is the name the cell, and *cell_type* is typically one of values that are mapped to the FormalPro PA types in formalpro.ini. The default cell_type strings in formalpro.ini are:

CLRFF	CLHRFF	CFRFF
ALRLA	AFRFF	AHRLA
LSHIFTER	ISOCELL	NONE

Examples

Assume there is a library cell named “cell2” that implements clock low retention flip-flop behavior. This statement assigns the attribute string “CLRFF” to all instances of cell2:

```
CLRFF LIBCELL, cell2
```

You can specify a *cell_type* that is not one of the default strings, but you must then override the mapping in the formalpro.ini file. For example, this statement assigns the attribute string “FOO” to instances of cell2:

```
FOO LIBCELL, cell2
```

This changes the attribute string associated with the FormalPro PA type CLRFF to “FOO”

```
-paConfigCLRFF FOO
```

Assigning Cell Type Attribute Strings Using a UPF File

The preferred way that FormalPro gets the cell type attribute string is from a UPF file, not a configuration file (use the -UPF option, instead of the -paConfig option). The UPF command below has the same affect as the configuration file statement.

UPF syntax:

```
map_retention_cell RET_FIFO -domain PD_FIFO -lib_cell_type CLRFF
-lib_model_name cell2 -port SAVE save_fifo -port NRESTORE nrestore_fifo
-port TVDD VDD -port VDD VDD_pri
```

Configuration file statement:

```
CLRFF LIBCELL, cell2
```

For details, refer to the UPF Standards document the syntax of the following commands: map_retention_cell, map_level_shifter, and map_isolation_cell.

GUI Access

Location: **Options** dialog box —
A specific pane > **Control** tab >
B specific pane > **Control** tab >
Power Aware field

Action: Enter the filename, or select it using the browser.

-paConfig<pa_type>

Design-specific

Alias: None

Maps *cell_type*, attribute string provided by a UPF file or a configuration file, to a FormalPro power-aware type.

In previous releases, this option was named pmConfig<pm_type>.

Usage

- paConfigCHRFF *cell_type*

  cell_type is a string assigned to clock high retention flip flops.

- -paConfigCLRFF *cell_type*

cell_type is a string assigned to clock low retention flip flops.

- -paConfigCFRFF *cell_type*

  cell_type is a string assigned to clock free retention flip flops.

- -paConfigAHRLA *cell_type*

  cell_type is a string assigned to active high retention latches.

- -paConfigALRLA *cell_type*

  cell_type is a string assigned to active low retention latches.

- -paConfigAFRLA *cell_type*

  cell_type is a string assigned to active free retention latches.

- -paConfigLSHIFTER *cell_type*

  cell_type is a string assigned to level shifters.

- -paConfigISOCELL *cell_type*

  cell_type is a string assigned to isolation cells.

- -paConfigNONE *cell_type*

  cell_type is a string used for objects with no PA attribute but that are matched with an object that has a PA attribute. Used internally by the PA checker.

Description

Use this option to map a cell_type attribute (string) to a FormalPro PA type. The PA checker uses the attribute to identify, and report on, PA objects generated in the compile phase. Use this option when information in a UPF file (or a Formalpro configuration file) assigns an unexpected attribute string to a given type of PA type.

By default, attribute strings are mapped to PA types in formalpro.ini. This option overrides the mapping specified in formalpro.ini.

Examples

Suppose that in a UPF file named “my_upf”, the cell type attribute string “clk_low_ret_ff” is assigned to a library cell “cell2”:

```
map_retention_cell RET_FIFO -domain PD_FIFO -lib_cell_type clk_low_ret_ff
-lib_model_name cell2 -port SAVE save_fifo -port NRESTORE nrestore_fifo
-port TVDD VDD -port VDD VDD_pri
```

The following maps “clk_low_ret_ff” the PA type CLRFF:

```
formalpro -b -UPF my_upf -paConfigCLRFF clk_low_ret_ff
```

Suppose that in a configuration file named “my_cfg_file”, the cell type attribute string “clow_ret_ff” is assigned to a library cell “cell9”:

```
clow_ret_ff LIBCELL, cell9
```

The following maps “clk_low_ret_ff” the PA type CLRFF:

```
formalpro -b -paConfigfile my_cfg_file \
-paConfigCLRFF clow_ret_ff
```

-paLib<pa_type>

Design-specific

Alias: None

Maps *cell_type*, an attribute string provided by a Liberty or Verilog library, to a FormalPro PA type.

In previous releases, this option was named pmLib<pm_type>.

Usage

- -paLibCHRFF *cell_type*

  cell_type is a string assigned clock high retention flip flops.

- -paLibCLRFF *cell_type*

  cell_type is a string assigned clock low retention flip flops.

- -paLibCFRFF *cell_type*

  cell_type is a string assigned clock free retention flip flops.

- -paLibAHRLA *cell_type*

  cell_type is a string assigned active high retention latches.

- -paLibALRLA *cell_type*

  cell_type is a string assigned active low retention latches.

- -paLibAFRLA *cell_type*

  cell_type is a string assigned to active free retention latches.

Description

Use this option to map a cell_type attribute (string) that is defined in a Liberty library to a FormalPro PA type. The PA checker uses the attribute to identify and report on PA objects generated in the compile phase.

Use this option when information in a Liberty library assigns an unexpected attribute string to a given PA type.

By default, attribute strings are mapped to PA types in formalpro.ini. This option overrides the mapping specified in formalpro.ini.

Supported Attributes

LIBERTY	VERILOG
is_isolation _cell : true ;	// pragma is_isolation _cell
is_level_shifter : true ;	// pragma is_level_shifter
power_gating_cell : “type_1”;	// pragma power_gating_cell : “type_1”
retention_cell : “type_1”;	// pragma retention_cell : “type_1”
switch_cell_type : coarse_grain;	// pragma switch_cell_type : coarse_grain
always_on : true ;	// pragma always_on

Examples

Suppose a Liberty library contains a definition of a clock low retention cell that assigns the cell type attribute string “foo”:

```
formalpro -b -paLibCLRFF foo -slib gate_liberty.lib
```

-propagateDontCare

Design-specific

Alias: None

-Controls how FormalPro handles don’t care situations when compiling an RTL design.

Usage

- -propagateDontCare {all | none}

  all — enables this functionality (default setting for design A).

  none — disables this functionality (default setting for design B).

You **can not** specify the following combination:

```
formalpro \
   -a -propagatedontcare none designfile \
   -b -propagatedontcare all designfile
```

Description

This switch is used primarily for RTL-to-gate comparisons, so under normal circumstances, it is only neccesary for the RTL design (design A). If you need to specify this switch for both designs, you can include it in the -common scope.

An error is returned if you attempt to specify “none” for design A and “all” for design “B”.

GUI Access

Location: **Options** dialog box —
A specific pane > **RTL** tab >
B specific pane > **RTL** tab >
Propagate dontcare dropdown box

Action: Select argument.

Examples

```
formalpro -a ./rtl/design_a.v \
                -propagatedontcare none \
            -b ./gate/design_b.v
```

-partialSumCheck

Global

Alias: None

Enables the verification of DW02_multp and DW02_tree modules.

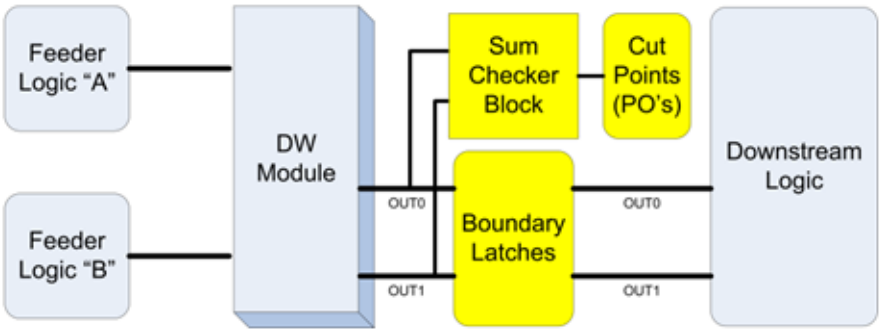
Usage

- -partialSumCheck — Enabled.
- -noPartialSumCheck — Disabled. Default.

Description

DW02_multp and DW02_tree modules contain partial sum outputs that cannot be verified by directly comparing RTL models to the gates implementation. These modules can be verified by adding the two partial sums together and checking the results between the two designs.

This option inserts an adder circuit directly into the DW02_multp or DW02_tree module after it is read in and can be seen in the design schematic as highlighted in yellow below.



The outputs of the modules are isolated from the downstream logic with latches that provide a new cut point for verification. The addition of summing compare points (names) provides verification that the RTL and the revised DW modules are equivalent.

The addition of latches breaks the DW module away from the following circuits and provides primary inputs for the down-stream logic. Care is taken to propagate don't-care functions thru these latches to simplify the verification process and improve performance.

GUI Access

Location: **Options** dialog box —
General pane > **Compile** tab >
Enable Partial Sum Check checkbox

Action: Use checkbox to enable or disable.

Examples

```
formalpro -partialSumCheck
   -a -rtl ./rtl/dw_uns_6x6.v -b ./gate/dw_uns_6x6.v
```

If an error is found in the gate-level implementation of the DW02_multp or DW02_tree, FormalPro reports differences on the Sum block output nets, which are named "sumchk_<bit

number>". In the formalpro.cache/reports/detailedComparison.report file, they are reported as in-equivalent targets:

```
5. solved: in-equivalent targets.
-----------------------------------
#
#
to76
\A.dw_uns_6x6.DUT_TREE.__mgc_fv_sum_checker.__mgc_fv_sumBlkBB.sumchk_11
\B.dw_uns_6x6.DUT_TREE.__mgc_fv_sum_checker.__mgc_fv_sumBlkBB.sumchk_11
#
#
to77
\A.dw_uns_6x6.DUT_TREE.__mgc_fv_sum_checker.__mgc_fv_sumBlkBB.sumchk_12
\B.dw_uns_6x6.DUT_TREE.__mgc_fv_sum_checker.__mgc_fv_sumBlkBB.sumchk_12
#
#
to78
\A.dw_uns_6x6.DUT_TREE.__mgc_fv_sum_checker.__mgc_fv_sumBlkBB.sumchk_13
\B.dw_uns_6x6.DUT_TREE.__mgc_fv_sum_checker.__mgc_fv_sumBlkBB.sumchk_13

The boundary latches that are inserted have the name of the output ports
embedded in the name. For example, for DW02_multp, which has ports "out0"
and "out1", the boundary latches are named:
   ""__mgc_fv_dl_out0_<bit_number>" and
   ""__mgc_fv_dl_out1_<bit_number>"
```

-pruneMuxAheadOfLatch

Design-specific

Alias: None

Removes redundant 2x1 multiplexers just ahead of latches during compilation.

Usage

- -pruneMuxAheadOfLatch — enables this functionality. Default.
- -nopruneMuxAheadOfLatch — disables this functionality.

GUI Access

Location: **Options** dialog box —
A specific pane > **RTL** tab >
B specific pane > **RTL** tab >
Prune mux ahead of latch check box

Action: Use check box to disable/enable.

Examples

The following example disables the *pruneMuxAheadOfLatch* functionality:

```
formalpro -nopruneMuxAheadOfLatch \
```

-QQbarMerge

Global

Alias: None

Compiles library files containing two registers representing Q and Qbar ports so that the registers are represented as a single DFF.

Usage

- -QQbarMerge — enables this functionality (default).
- -noQQbarMerge — disables this functionality.

Description

This switch allows you to compile your RTL design in the same way you synthesized your design.

GUI Access

Location:	**Options** dialog box — **General** pane > **Lib** tab
Action:	Enable: **Select Merge Q Q-bar** Disable: **Unselect Merge Q Q-bar**

Examples

```
formalpro -qqbarmerge \
            -a ./rtl/design_a.v -b ./gate/design_b.v
```

-QQbarSetResetMerge

Global

Alias: None

Compiles library files containing set- or reset-dominant registers defined with two UDPs so that the registers are represented as a single DFF.

Usage

- -QQbarSetResetMerge — enables this functionality (default).
- -noQQbarSetResetMerge — disables this functionality.

Description

This switch allows you to compile your RTL design in the same way you synthesized your design.

GUI Access

Location:	**Options** dialog box —
	General pane > **Lib** tab
Action:	Enable: Select **Merge Q Q-bar set/reset**
	Disable: Unselect **Merge Q Q-bar set/reset**

Examples

```
formalpro -qqbarsetresetmerge \
            -a ./rtl/design_a.v -b ./gate/design_b.v
```

-queueLicense

Global

Alias: None

Enables/Disables the FLEXlm license queuing feature.

Usage

- -queueLicense — enables license queuing (default).
- -noqueueLicense — disables license queuing.

Description

By default, when you invoke FormalPro and a license is not immediately available, a request is placed in a license queue. Once your request is in the license queue, FormalPro continuously monitors the license server for an available license. When a license becomes available, FormalPro automatically acquires it and continues executing. While in the queue, a pending message displays. Enter Ctrl-c from a shell prompt or click **Stop** on the GUI toolbar to exit the license queue at any time.

Note: A valid license must be available, or a license failure message displays.

GUI Access

Location:	**Options** dialog box —
	General pane > **Control** tab
Action:	Enable: Select **Enable license queuing**
	Disable: Unselect **Enable license queuing**

Examples

```
formalpro -queueLicense
```

-redundantRegMerge

Global

Alias: None

Compiles library files containing two UDPs that produce equal output so that the registers are represented as a single DFF.

Usage

- -redundantRegMerge — enables this functionality (default).
- -noredundantRegMerge — disables this functionality.

Description

This switch allows you to compile your RTL design in the same way you synthesized your design.

GUI Access

Location:	**Options** dialog box —
	General pane > **Lib** tab
Action:	Enable: **Select Merge Redundant register**
	Disable: **Unselect Merge Redundant register**

Examples

```
formalpro -redundantregmerge \
            -a ./rtl/design_a.v -b ./gate/design_b.v
```

-removeIgnoredOutputs

Global

Alias: None

Instructs FormalPro to ignore registers that 1) lead directly to a primary output and 2) are part of a net that you have constrained as ignorable.

Usage

- -removeIgnoredOutputs — enables this functionality.
- -noremoveIgnoredOutputs — disables this functionality (default).

Description

This switch is most useful when performing a pre- to post-scan verification.

For example, you should specify -removeIgnoredOutputs if you ignore the output of a scan chain and your scan-insertion tool places a shadow latch at the end of the scan chain. This prevents FormalPro from reporting an unmatched latch.

GUI Access

Location:	**Options** dialog box — **General** pane > Match tab
Action:	Enable: Select **Remove ignored output** Disable: Unselect **Remove ignored output**

Examples

```
formalpro -removeignoredoutputs \
            -a design_a.v \
            -b design_b.v
```

-reportUnmatchedDiffs

Global

Alias: None

Reports fed-by-unmatched targets that cannot be proven equivalent as “differences” instead of as removed targets.

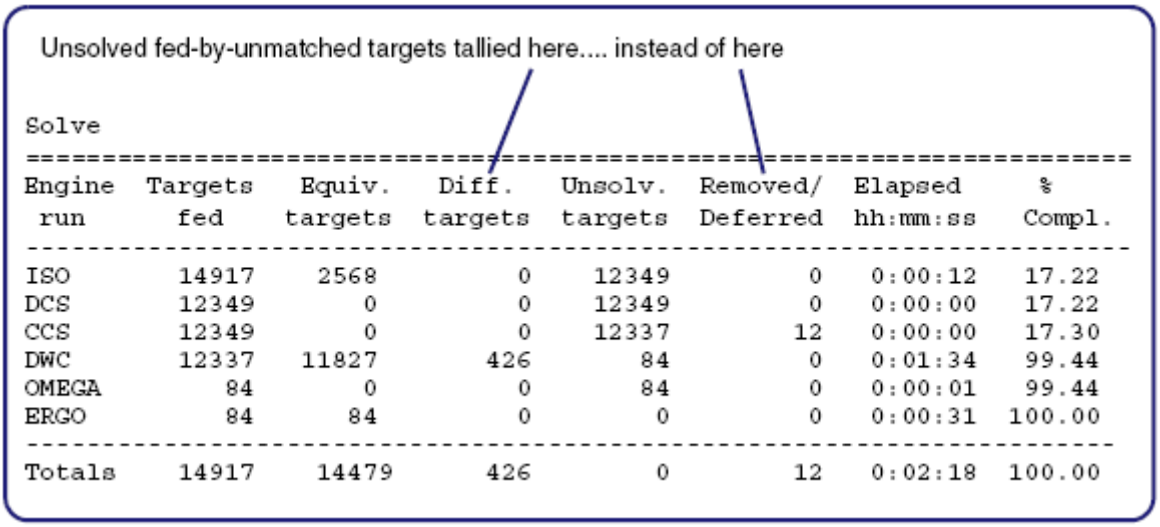
Usage

- -reportUnmatchedDiffs — reports targets as “differences” if they are fed by unmatched inputs and if they cannot be proven equivalent.
- -noreportUnmatchedDiffs — removes targets if they cannot be proven equivalent and if they are fed by unmatched inputs (default).

Description

Use this option to report targets as “differences” if they are fed by unmatched inputs and if they cannot be proven equivalent. By default, fed-by-unmatched targets that cannot be proven equivalent are “removed”.

Figure 2-2. Effects of -reportUnmatchedDiffs



Solve

Engine run	Targets fed	Equiv. targets	Diff. targets	Unsolv. targets	Removed/ Deferred	Elapsed hh:mm:ss	% Compl.
ISO	14917	2568	0	12349	0	0:00:12	17.22
DCS	12349	0	0	12349	0	0:00:00	17.22
CCS	12349	0	0	12337	12	0:00:00	17.30
DWC	12337	11827	426	84	0	0:01:34	99.44
OMEGA	84	0	0	84	0	0:00:01	99.44
ERGO	84	84	0	0	0	0:00:31	100.00
Totals	14917	14479	426	0	12	0:02:18	100.00

Note

By default, FormalPro attempts to solve fed-by-unmatched targets. However, if you specify -nosolveFedByUnmatched, no attempt is made to solve fed-by-unmatched targets. In this case, all fed-by-unmatched targets are reported as “removed”.

GUI Access

Command-line only.

Examples

```
formalpro -reportUnmatchedDiff \
            -a design_a.v \
            -b design_b.v
```

-reports

Stand-alone

Alias: None

Completes the generation of reports if the run did not complete.

Usage

- -reports

Description

Completes the detailed comparison, comparison summary, and run statistics reports from a run that was interrupted by pressing Ctrl-C during the solve stage.

GUI Access

Location: **Reports** > **Generate reports** menu item

Examples

```
formalpro -reports
```

-restart

Global

Alias: None

Restarts a verification run at a specified stage.

Usage

- -restart {a | b | match | constraint | solve | coverage}

  a | b — restarts at the compile stage and compiles only the specified design-side, either A or B. For example, *-restart a* compiles the A-side source and leaves the B-side compiled from the previous run.

  match — restarts at the beginning of the match stage. Use this option when you make changes to the rule file or match file.

  constraint — restarts at the beginning of the match stage or compile stage, depending on the types of changes in the constraint file. Use this option when you make changes to the constraint file.

solve — restarts at the beginning of the solve stage. When this option is used to rerun the solve phase, the results of the previous solve phase are preserved and any additional resolved targets are included in the results.

coverage — restarts an aborted or timed-out solve phase to complete the coverage analysis. Use this when an aborted solve phase results in an incomplete or missing The verification coverage report.

A restarted coverage function spends up to 60 seconds per unsolved target and can timeout depending on the specified solve-time. Before restarting the coverage function, determine the number of unsolved targets with the Detailed Comparison report, and set the solve-time limits accordingly. See -solveTimeLimit and -strategy.

Note

The -restart option leaves the existing cache in tact and any new output files created during the run overwrite the corresponding files in the existing cache.

Description

To restart a previous run, add the *-restart* option to the command used for the initial invocation.

You can also use the -resume option to restart/resume a verification run in the solve stage. The *-restart solve* and *-resume options* are similar except *-restart solve* restarts the solve stage at the beginning sequence of the solve algorithms. The *-resume* restarts the solve phase at the point in algorithm sequence where the run was stopped and progresses forward into the increasingly intensive algorithms. In a design where simple targets may have been blocked by the presence of large cone targets, like multipliers, you should use the *-restart solve* option. You can monitor the progress of the solve stage with the -logLevel *full* option.

Caution

Use the appropriate restart option! FormalPro does not verify that modifications are included in a restarted run. For example, if you modify a design source but use *-restart match* instead of restarting at the compile stage, the run proceeds using the last completed compilation. The modifications are not included in the run; no warning is issued.

GUI Access

Location: Toolbar > 

Action: 1. Select the stage from the dropdown box.

2. Click  to execute the run.

Examples

The following example uses the -restart and -matchfile switches with the initial invocation command to add an explicit match file and restart the verification run at the match stage:

```
formalpro -restart match -matchfile match.cmd \
            -a design_a.v \
            -b design_b.v
```

-resume

Global

Alias: None

Resumes a verification run in the solve stage.

Usage

- -resume

Description

Depending on the verification run status, the *-resume* option reruns the solve stage or resumes the solve stage where it left off. The *-resume* option can be used in any of the following ways to resolve remaining verification targets:

- **Repeat the solve phase** — rerun the solve phase for a completed verification run. When *-resume* is used to rerun the solve phase, the results of the previous solve phase are preserved and any additional resolved targets are included in the results.
- **Increase solve effort** — in conjunction with the --strategy option, resume the solve stage and increase the amount of solve effort used.
- **Increase solve time** — in conjunction with the --solveTimeLimit option, resume the solve stage and increase the time allowed for the verification run.

You can also use the --restart option to restart/resume a verification run at the beginning of the compile, match, or solve stage. The *-restart solve* and *-resume options* are similar except *-restart solve* restarts the solve stage at the beginning sequence of the solve algorithms. The *-resume* restarts the solve phase at the point in algorithm sequence where the run was stopped and progresses forward into the increasingly intensive algorithms. In a design where simple targets may have been blocked by the presence of large cone targets, like multipliers, you should use the *-restart solve* option. You can monitor the progress of the solve stage with the -logLevel *full* option.

GUI Access

Location: Toolbar > 

Action: 1. Click  to execute the run.

Examples

```
formalpro   -strategy HIGH -resume
            -a ./rtl/design_a.vhd \
            -b ./gate/design_b.v
```

-retime

Global

Alias: None

Enables the retime solver named HAMBLE in the *solve.log* trace. This is activated when a retimed register that is not paired A/B with another is detected in the fanin of another target.

Usage

- -retime — enables the retime solver
- -noRetime — disables the retime solver (default)

Description

The sequential solver engine is activated when -retime is true. The function relies on proper matching to provide targets where the fanin includes non-equivalent or unmatched register pairs and their cones. The targets being fed are matched properly and expected to be A/B functionally equivalent, except when the fanin contains a retimed circuit. If a difference is detected in a likely retimed fanin, the debugger highlights the retimed and non-retimed logic differences without distinction. It might be difficult to decipher the differentiated output. To resolve the correctness, use sequential simulation.

Simple retiming for speed and area can be solved for circuits and when datapaths are less than 20 bits wide.

Note

The initial application is for Precision FPGA compilation of Xilinx Vx5 devices including inferred RAM and DSP elements. Please see -flow retime for this support.

Examples

```
formalpro -retime {any FormalPro command follows}
```

-rtl

Design-specific

Alias: None

Specifies that the files within a design scope are RTL.

Usage

- -rtl

Description

FormalPro, by default, parses your design files to determine if they are RTL or gate-level. This switch explicitly informs FormalPro that a design is RTL.

GUI Access

Location: **Options** dialog box —
A specific pane > **Control** tab >
B specific pane > **Control** tab >
Design level dropdown box

Action: Select rtl

Examples

```
formalpro -a -rtl ./rtl/design_a.vhd \
          -b ./gate/design_b.v
```

-rtlIgnoreNoPathBBIns

Design-specific

Alias: None

Instructs FormalPro to ignore targets created for input ports for an empty module.

Usage

- -rtlIgnoreNoPathBBIns — enables this functionality (default for flow files for Cadence, DC Ultra, Precision, and Oasys-RTL)

- -NortlIgnoreNoPathBBIns — disables this functionality (default for gate to gate tests and generic FormalPro command line)

Description

This option ignores the targets created for input ports for an empty module if there are no output ports present for the same module. This avoids unmatched output (black box input) ports if the empty port has been optimized in the reference design. Use this option to ignore simulation related modules that are not synthesizable.

GUI Access

Location: **Options** dialog box —
A specific pane > **RTL** tab
Ignore NoPath Blackbox Inputs check box
B specific pane > **RTL** tab
Ignore NoPath Blackbox Inputs check box

Action: Use check box to disable or enable.

Examples

In the following example, FormalPro ignores the targets created for input ports for empty modules in the reference design.

```
formalpro -a -rtlIgnoreNoPathBBIns ./rtl/design_a.vhd \
          -b ./gate/design_b.v
```

-rtlIgnoreVHDLComponentError

Global

Alias: None

Relaxes the VHDL language requirements for interfacing VHDL to a Verilog module as a component.

Usage

- -rtlIgnoreVHDLComponentError — Uses relaxed VHDL language requirements.
- -noRtlIgnoreVHDLComponentError — Uses stricter VHDL language requirements.

GUI Access

Location: **Options** dialog box —
A specific pane > **RTL** tab >
B specific pane > **RTL** tab >
RTL Compiler Settings panel

Action: Select Disable component parameter checks

-rtlMemoryLimit

Design-specific

Alias: None

Specifies an upper limit for the size of a memory module for FormalPro to compile in RTL designs.

Usage

- -rtlMemoryLimit [integer]

  integer — The default value of integer is -1 (disable), which refers to the number of memory elements in the array.

Description

This switch is a register count threshold, which automatically black boxes RTL modules that FormalPro determines to be memory modules. Memories that are greater than the integer value you specify will be made a black box by the RTL compiler.

GUI Access

Location: **Options** dialog box —
A specific pane > **RTL** tab
B specific pane > **RTL** tab
Size limit dropdown box

Action: A) Type value or
B) Select argument

Examples

Based on the following example, FormalPro will black box any memory modules that contain more than 2048 memory elements. This would include, for example, a 128 x 32 memory, where the address is 5 bits, which contains 4096 bits.

```
formalpro -a -rtlMemoryLimit 2048 ./rtl/design_a.vhd \
          -b ./gate/design_b.v
```

-rtlSimWarnings

Global

Alias: none

Promotes warnings related to simulation/synthesis miss-match due to RTL coding style. Copies these warnings from the compileDetails log to the main log. Use this option to improve the RTL.

Usage

- -rtlSimWarnings — Promotes warnings.
- -noRtlSimWarnings — Does not promote warnings.

GUI Access

Location: **Options** dialog box —
A specific pane > **RTL** tab >
B specific pane > **RTL** tab >
RTL Compiler Settings panel

Action: Select Allow Simulation Warnings

-rtlTreatDeclAsassign

Design-specific

Alias: None

Treats all Verilog/SV variable declarations with non-static initial values as continuous assignment statements during compilation.

Usage

- -rtlTreatDeclAsassign — enables this functionality.
- -nortlTreatDeclAsassign — disables this functionality. Default.

Description

This option is enabled in the Oasys-RTL flow for compatibility.

GUI Access

Location: **Options** dialog box —
A specific pane > **RTL** tab >
B specific pane > **RTL** tab >
Treat declaration as assignment check box

Action: Use check box to disable/enable.

Examples

The following example disables the rtlTreatDeclAsassign functionality:

```
formalpro -nortlTreatDeclAsassign \
```

RTL Naming Control

Design-specific

Alias: None

Controls how FormalPro names objects during the compilation of your RTL design files.

Usage

FormalPro applies the following switches to specify a particular object is either a scalar or vector register (DFF, latch, or tri-state).

- -dffInstScalarFormat “value” — default setting: %s_reg
- -dffInstVectorFormat “value” — default setting: %s_reg(%d)
- -dffInstMemoryFormat “value” — default setting: %s_reg(%d)(%d)
- -latchInstScalarFormat “value” — default setting: %s_lat
- -latchInstVectorFormat “value” — default setting: %s_lat(%d)
- -latchInstMemoryFormat “value” — default setting: %s_lat(%d)(%d)

FormalPro rarely applies the following switch, which is used in case two names are exactly the same.

- -arrayNameFormat “value” — default setting: **%s(%d)**

This switch applies to objects in an array. A possible change to value is: “%s[%d]”

- -recordNameFormat “value” — default setting: **%s.%s**

  This switch controls the naming style for RTL objects related to records. A possible change to value is “%s_%s”.

- -collisionNameFormat “value” — default setting: **%s_%d**

Syntax Requirements:

- Be sure to enclose value within quotation marks (“) when specifying any of these switches on the command line.
- You should specify an occurrence of “%s” before you specify a “%d”. If you have to swap the order, the value must appear similar to the following:

  ```
  -dffInstVectorFormat "%2\$d_text_%1\$s"
  ```

  which is similar in format to the fprintf UNIX command, where the %1 and %2 move the relative variables to the desired places.

Description

By altering these switches, you can increase the efficiency of the Match stage, because FormalPro will be able to complete more matches based on the canonical names. To do this successfully, you must know the naming style for your synthesis tool.

GUI Access

Location: **Options** dialog box —
A specific pane > **RTL names** tab
B specific pane > **RTL names** tab

Examples

Assume your netlist contains named register arrays in the following format (as would be seen in the Match Tool):

```
f + \B.top.foo_reg[1_7]
```

noting your synthesis tool surrounds array values by square brackets ([ ]) and separates them by an underscore (_). However, FormalPro compiles your RTL to appear as:

```
f + \A.top.foo(1)_reg(7)
```

FormalPro then compiles your RTL design to match the naming style in your netlist.

-ruleFile

Scope: Global

Alias: None

Specifies the location of a rule file containing implicit match rules.

Usage

- -ruleFile filename — specifies the location of the rule file.

  filename — species a file pathname. Non-literal pathnames are relative to the current directory.

Description

FormalPro automatically loads a default rule file that contains several pre-created rules that aid in matching the comparison points between your designs. The -ruleFile option lets you override the default rule file by specifying a different rule file.

Note

Before using this option you should have a full understanding of the operation and interaction of the default rule file, rule sets, the -ruleFile option and the -addRule file option. For this information, see the section titled “Rule Files” on page 207.

GUI Access

Location:	**Project** tab > **General** tab > Match rules entry box
Action:	Type the path to your rule file or use: 

Examples

```
formalpro -rulefile ./setup/rule.cmd \
            -a design_a.v \
            -b design_b.v
```

-slib, -slibF

Scope: Design-specific

Alias: None

Specifies Synopsys Liberty technology library files to use for design compilation.

Usage

- -slib *libFile* —specifies a Synopsys Liberty technology library.

libFile — a single library file. One libFile argument per switch.

- -slibF *libList* — specifies file containing a list of library files.

  libList — a file containing a list of library files . One libList argument per switch.

Description

Use these options for specifying a library file (-slib) or a list of library files (-slibF) to use for design compilation.

The format of libList file is shown in the following example:

```
# commented line
./lib/syn_library_1.lib          # a single Synopsys library file
./lib/syn_lib_*.lib              # wildcards are allowed
```

You can use -slib and -slibF multiple times on the command line.

GUI Access

Location:	**Project** tab > **A** tab
	Project tab > **B** tab
	Project tab > **Common** tab
Action:	Specifying a library file:
	1. Type the path to your library or use: 
	2. Select slib from the dropdown menu to the left of the library.
	Specifying a library file list:
	1. Type the path to your file list or use: 
	2. Select slibF from the dropdown menu to the left of the file.

Examples

```
formalpro -a designA.v \
            -b designB.v \
            -common -slib ./syn_lib_1.lib
```

-simplifyPipelineRegs

Scope: global

Alias: None

Eliminates false differences caused by offsetting inverters in pipelined registers.

Usage

- -simplifyPipelineRegs — enables this functionality.
- -nosimplifyPipelineRegs — disables this functionality (default).

Description

For the purpose of re-timing a design, inverters are sometimes added to a simple, straight-through pipeline of registers. This can result in FormalPro reporting a difference even though the additional inverters offset each other.

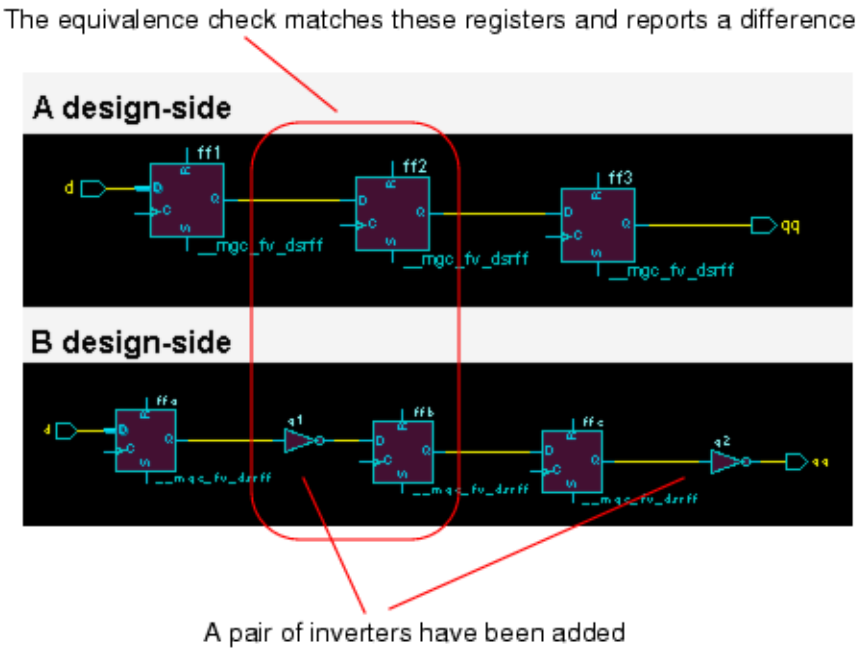
Figure 2-3 shows an example. The B-side schematic shows pipelined registers with two inverters. The two inverters offset each other, so the overall logic of the pipeline is unchanged. However, because the logic feeding the A-side register differs from the logic feeding the matching B-side register, the equivalence check reports a difference.

If upon examining a reported difference you find this situation, enable the -simplifyPipelineRegs switch.

Note

This switch resolves the problem by making the appropriate matched register pairs transparent, so you will see a reduction in register count and target count.

Figure 2-3. Inverters added to change timing



GUI Access

Location: **Options** dialog box —
General pane > **Options** > **Match** tab

Action: Enable: **Select Simplify pipeline registers**
Disable: Unselect **Simplify pipeline registers**

Examples

```
formalpro -simplifyPipelineRegs -a design_a.v \
          -b design_b.v
```

-solveFedByUnmatched

Scope: Global

Alias: None

Instructs FormalPro to solve any targets that have an unmatched input.

Usage

- -solveFedByUnmatched — enables this functionality (default).
- -nosolveFedByUnmatched — disables this functionality.

Description

When FormalPro encounters an unmatched input, it marks all targets in the fan-out of that input as “fed by unmatched”. By default, FormalPro attempts to solve these targets, but you can override this functionality by specifying -nosolveFedByUnmatched. In this case, the fed-by-unmatched are removed.

GUI Access

Location: **Options** dialog box —
General pane > **Solve** tab

Action: Enable: **Select Solve targets fed by unmatched**
Disable: **Unselect Solve targets fed by unmatched**

Examples

```
formalpro -solvefedbyunmatched \
            -a design_a.v \
            -b design_b.v
```

-solveOrder

Scope: Global

Alias: None

Applies a control file for the solve engine sequence.

Usage

- -solveOrder *controlFile* — specifies a factory-provided control file.

Description

This switch applies a control file supplied by the factory to correct a solve issue for a specific application. This is a control file for the solve engine sequence. If there is an issue with solving, the factory can create a revision to this file, and it can be applied from the command line.

This is an expert level command, and there is no user-configurable text in the solveOrder files.

Examples

```
formalpro -solveOrder DSP2\
```

-solveTimeLimit

Scope: Global

Alias: None

Sets a specific timeout value for the solve stage, overriding the value set by -strategy switch.

Usage

- -solveTimeLimit {<minutes> | <hours>:<minutes>}

Description

Specifies the maximum duration of the solve stage before timing out with unsolved targets. Once the time limit is reached, the process continues for a short time until it reaches a checkpoint, where it can exit cleanly. In rare cases, it may take thirty minutes or more to reach a checkpoint.

The -solveTimeLimit switch overrides the time limit set by the --strategy switch. If you do not specify -solveTimeLimit on the command line, (or if in the GUI you set the time limit to 0 or infinity), the time limit reverts to setting defined by -strategy (default, four hours).

The -solveTimeLimit switch is useful in batch-mode scripts that invoke multiple FormalPro runs, to ensure that a script run overnight does not get bogged down in the solve stages of one or two designs.

GUI Access

Location: **Options** dialog box —

General pane > **Solve** tab

Action: Enable: Enter a non-zero value in text field to over-ride the time limit determined by the --strategy setting.

Disable: Enter 0 or delete the value in the text entry field (this resets the text field to “infinity” and removes -solveTimeLimit from the command line.

Examples

The following examples show two different formats for setting the solve time limit to 1 hour and 30 minutes:

```
formalpro -solvetimelimit 90 -a design_a.v -b design_b.v

formalpro -solvetimelimit 1:30 -a design_a.v -b design_b.v
```

-stopAfter

Scope: Global

Alias: None

Stops the run after the specified phase.

Usage

- -stopAfter {compile | match}

Description

Stops the run either after the compile phase (before the start of the match phase) or after the match phase (before the start of the solve phase).

GUI Access

None.

Examples

```
formalpro -stopAfter compile\
            -a design_a.v \
            -b design_b.v
```

-stopOnBlackBox

Scope: Global

Alias: -sobb or -nosobb

Instructs FormalPro to end a run if it black boxes a module or entity that was not user-specified.

Usage

- -stopOnBlackBox — enables this functionality.
- -nostopOnBlackBox — disables this functionality (default).

Description

The FormalPro run ends after the compile stage when stopping a run due to a non-user-specified black box, and the following message is issued:

```
ERROR: design <A|B> contains black boxes
```

GUI Access

Location:	**Options** dialog box —
	General pane > **Control** tab
Action:	Enable: Select **Stop on black box**
	Disable: Unselect **Stop on black box**

Examples

```
formalpro -sobb \
            -a design_a.v \
            -b design_b.v
```

-stopOnConfigError

Scope: Global

Alias: None

Stops the run when encountering an error in a configuration file.

Usage

- -stopOnConfigError — enables this functionality (default).
- -nostopOnConfigError — disables this functionality.

Description

StopOnConfigError stops the run at the end of the compilation phase when an error is encountered in a configuration file. If -nostopOnConfigError is used, a warning is issued and the run continues.

GUI Access

Location:	**Options** dialog box —
	General pane > **Control** tab
Action:	Enable: Select **Stop on config error**
	Disable: Unselect **Stop on config error**

Examples

```
formalpro -nostoponconfigerror \
            -a design_a.v \
            -b design_b.v
```

-stopOnConstraintError

Scope: Global

Alias: None

Ends the run when an error is encountered in the constraint file or explicit match file.

Usage

- -stopOnConstraintError — enables this functionality (default).
- -nostopOnConstraintError — disables this functionality.

Description

By default, errors in the constraint file or explicitly match file forces FormalPro to end after the Match stage with an error statement. However, when you specify -nostopOnConstraintError, FormalPro prints all the errors as warnings and continues beyond the matching step.

The following is an example of how the Warning might appear:

```
WARNING(file:./setup/constraints line:4) '\A.top.I2.counte_reg(0)' is not
a valid register name - command ignored.
```

GUI Access

Location:	**Options** dialog box — **General** pane > **Control** tab
Action:	Enable: Select **Stop on constraint error** Disable: Unselect **Stop on constraint error**

Examples

```
formalpro -nostoponconstrainterror \
            -a design_a.v \
            -b design_b.v
```

-stopOnCycles

Scope: Global

Alias: -socyc

Instructs FormalPro to end the run during the solve stage when combinational cycles are detected in the design.

Usage

- -stopOnCycles — enables this functionality.
- -nostopOnCycles — disables this functionality (default).

Description

FormalPro will stop the verification during the Solve stage (after completing the Isomorphism engine) when an active cycle exists in your design.

This switch also disables the automatic solving of targets fed by combinational cycles, which is controlled by --cycleSolve.

GUI Access

Location:	**Options** dialog box — **General** pane > **Control** tab
Action:	Enable: Select **Stop on cycles** Disable: Unselect **Stop on cycles**

Examples

```
formalpro -stoponcycles \
            -a design_a.v \
            -b design_b.v
```

-stopOnDiff

Scope: Global

Alias: -sod

Instructs FormalPro to end a run if it encounters a specified number of differences.

Usage

- -stopOnDiff integer

integer — a number greater than or equal to one. The default value is “infinity”.

Description

FormalPro by default solves all targets regardless of the number of differences found during the run. When you do specify this switch and FormalPro ends the run based on your setting, it issues the following information at the end of the solve stage:

```
Notice: the number of differences detected has exceeded the maximum.
```

GUI Access

Location:	**Options** dialog box — **General** pane > **Control** tab
Action:	Enable: 1. Select **Stop on difference** 2. Type value in dropdown box or use arrows to change Disable: Unselect **Stop on difference**

Examples

```
formalpro -stopondiff 15 \
            -a design_a.v \
            -b design_b.v
```

-stopOnMissing

Scope: Global

Alias: -somg or -nosomg

Instructs FormalPro to end a run if it encounters a missing library cell or design unit.

Usage

- -stopOnMissing — enables this functionality.
- -nostopOnMissing — disables this functionality (default).

Description

A FormalPro run ends after the compile stage when stopping a run due to a missing module, and the following message is issued:

```
ERROR: There are modules referenced but not defined in design <A|B>
```

GUI Access

Location: **Options** dialog box —
General pane > **Control** tab

Action: Enable: Select **Stop on missing**
Disable: Unselect **Stop on missing**

Examples

```
formalpro -stopOnMissing \
            -a design_a.v \
            -b design_b.v
```

-stopOnUnmatched

Scope: Global

Alias: -soum or -nosoum

Instructs FormalPro to end the run if any comparison points remain unmatched after applying all implicit match rules and explicit match statements.

Usage

- -stopOnUnmatched — enables this functionality.
- -nostopOnUnmatched — disables this functionality (default).

Description

A FormalPro run ends after the match stage when stopping a run due to an unmatched comparison point, and the following message is issued:

```
ERROR: unmatched ports or registers remain
```

FormalPro will not stop the run when encountering a benign unmatched comparison point.

Your log and report files could report unmatched registers, even though the run did not stop, if you specify --gatedClocks. The presence of this switch attempts to prune redundant registers from your designs.

GUI Access

Location: **Options** dialog box —
General pane > **Control** tab

Action: Enable: Select **Stop on unmatched**

Disable: Unselect **Stop on unmatched**

Examples

```
formalpro -soum \
            -a design_a.v \
            -b design_b.v
```

-strategy

Scope: Global

Alias: None

Specifies the effort expended in the solve stage before timing out.

Usage

- -strategy [low | medium | high | extreme]

  low — 1 hr

  medium — 4 hrs (default)

  high — 12 hrs

  extreme — until the solve-order sequence finishes. This is typically 26 to 36 solve engine increments, which can be observed with the -logLevel full option.

Description

This switch allows you to specify one of four settings that control the duration of the solve stage before timing out with unsolved targets. After a timeout, the solve status of all targets is logged and available for viewing.

Unsolved targets can sometimes be resolved by changing the strategy setting and resuming the run (see --resume).

To set a specific timeout value, see --solveTimeLimit.

GUI Access

Location: **Options** dialog box —

General pane > **Solve** tab > **Solve strategy** dropdown box

Action: Select argument.

Examples

```
formalpro -strategy high \
            -a design_a.v \
            -b design_b.v
```

Suffix Control Switches (Design Files)

Scope: Global

Alias: -svlog (-suffixVerilog), -svhdl (-suffixVHDL)

This group of switches allows you to specify the suffix styles used for your design files.

Usage

- -suffixVerilog extensionList — Specifies extensions used for Verilog design files. Default extensions: v, .V, .vg, .VG, .vo, .VO, .vlg, .VLG, .verilog, .VERILOG, .vqo, .VQO, .vm, .VM, .vqm, and .VQM
- -suffixSystemVerilog *extensionList* — Specifies extensions used for SystemVerilog files. Default extensions: sv, SV
- -suffixVHDL extensionList — Specifies valid extensions for VHDL design files. Default extensions: .vhd, .VHD, .hdl, .HDL, .vhdl, and .VHDL.
- -suffixEDIF extensionList — Specifies extensions used for EDIF files. Default extensions: .edf, .edif, .EDF, .EDIF, .EDN, .edn

  Where extensionList is a colon-delimited list of extensions.

Description

This group of switches allow you to override the default file extensions defined in the formalpro.ini and used by the FormalPro when compiling design files.

For example, the default formalpro.ini defines sv and SV as SystemVerilog suffixes. Using the -suffixSystemVerilog switch you can override the SystemVerilog suffixes defined in formalpro.ini, replacing them with the suffixes in extensionList.

Specify these switches in the global section of the command line. They apply to all design files (but not library files) the A- and B design.

Compressed Design Files

FormalPro supports compressed design files. The file must be a single compressed file (not multiple files combined during compression). It must be Verilog netlist — any file that needs to be processed by the RTL compiler cannot be in compressed format.

The supported compression types are:

.gz	Unix/PC gzip
.bz2	Unix bzip2
.Z	Unix compress
.ZIP .zip	Unix zip; PC WinZip, PKzip; Windows OS file compress

Compressed files must use one of the suffixes listed above. The compression suffix follows the language suffix. For example, a Verilog netlist file compressed by gzip might look like this: *filename.v.gz*.

GUI Access

Location:	**Options** dialog box — **General** pane > **Compile** tab
Action:	Type in space-separated list of allowable suffixes.

Examples

Assume that at your site you use the extension .vg for Verilog designs, in addition to .v and .V, which are the defaults. Given this assumption, you would use the -suffixVerilog switch as follows:

```
formalpro -suffixverilog .v:.V:.vg \
            -a design_a.v \
            -b design_b.v
```

Be sure that you specify all possible extensions as an argument to the switch, because all default extensions replaced by your entry.

You can use wildcards in your suffix arguments, as shown in the following example for -suffixVHDL:

```
formalpro -suffixvhdl .vhd:.vhdl:.vhd_*:.vhdl_* \
            -a design_a.vhd \
            -b design_b.vhd
```

This is useful if your files are subject to version control.

Suffix Control Switches (Library Files)

Scope: Global

Alias: None

Specifies suffix styles used for library files.

Usage

- -suffixVlogLib extensionList — Specifies extensions used for Verilog library files. Default extensions: .v, .V, .vg, .VG, .vo, and .VO
- -suffixSystemVlogLib *extensionList* — Specifies extensions used for System Verilog library files. Default extensions: .sv and .SV.
- -suffixDftLib extensionList — Specifies extensions used for Mentor Graphics FastScan ATPG files. Default extensions: .atpglib,. ATPGLIB, .lib and LIB
- -suffixSynLib extensionList — Specifies extensions used for Synopsys Liberty files. Default extensions: .slib, .SLIB, .lib and .LIB

Where extensionList is a colon-separated list of extensions.

Description

This group of switches allow you to override the default file extensions defined in the formalpro.ini and used by FormalPro when compiling a library file.

For example, the default formalpro.ini defines .v and .V as SystemVerilog suffixes. Using the **-suffixVlogLib** switch you can override the Verilog suffixes defined in formalpro.ini, replacing them with the suffixes in extensionList.

The switches go in the global section of the command line and they apply to all appropriate files specified in the command line's **-a**, **-b**, and **-common** scope.

Compressed Design Files

FormalPro supports compressed library files. The library must be a single compressed file (not multiple files combined during compression). It must be in Verilog or Liberty format. Any file that needs to be processed by the RTL compiler cannot be in compressed format.

The supported compression types are:

.gz	Unix/PC gzip
.bz2	Unix bzip2
.Z	Unix compress
.ZIP .zip	Unix zip; PC WinZip, PKzip; Windows OS file compress

The compressed file must use one of the suffixes listed above. The compression suffix follows the language suffix. For example, a Verilog library file compressed by gzip might look like this: *filename*.v.gz.

GUI Access

Location:	**Options** dialog box — Project **General** pane > **Lib** tab
Action:	Type a space-separated list of allowable suffixes.

Examples

Assume that you use the extension .vg for Verilog libraries, in addition to .v and .V, which are the defaults. Given this assumption, you would use the -suffixVlogLib switch as follows:

```
formalpro -suffixvloglib .v:.V:.vg \
            -a design_a.v \
            -b design_b.v -v lib.vg
```

Be sure that you specify all possible extensions as an argument to the switch, because the defaults will be overwritten by your entry.

You can also use wildcards in your suffix arguments, as shown in the following example for -suffixDftLib:

```
formalpro -suffixdftlib .atpglib:.atpglib_*: \
            -a design_a.v \
            -b design_b.v -alib lib.atpglib
```

This is useful if your files are subject to version control.

-suppress

Scope: Global

Alias: None

Suppresses specified error, warning, and information messages issued during the translate and compile phase.

Usage

- -suppress filename

  filename — error configuration file that specifies the types of errors to suppress. Use a copy of the directives in *formalpro.errcfg as a template to create this file.*

Description

By default, message suppression is based on the directives defined in *$FORMALPRO_HOME/lib/formalpro.errcfg*. If you use the *-suppress* option, the directives in the specified error configuration file are used instead.

To avoid the display of many undesired messages, you should start with a copy of the message directives in the *formalpro.errcfg* file, and edit it as needed to create a custom error configuration file.

Syntax for defining directives can be found in the comments of the *formalpro.errcfg* file. Note that the directive Usage requires the identification number included in the error/warning/information message when it displays.

GUI Access

Location:	**Project** tab > **General** tab > **Suppress File** entry box
Action:	Type the path to the custom error configuration file

Examples

The following example specifies the custom error configuration file *suppress.errcfg* to filter messages:

```
formalpro -suppress ./control/suppress.errcfg -a rtl/design_a.v \
           -b gate/design_b.v
```

-SV

Scope: Design-specific

Alias: sv2009

Compiles files with SystemVerilog file extensions as SystemVerilog 2009 files.

Usage

- -sv *filename* [*filename*]...

  filename — Specifies a SystemVerilog file.

Description

This switch must precede design files to which it applies. This switch applies to all files within a design scope (-a, -b, or -common) including files specified in a file list or by any other means. See the examples below.

By default (when FormalPro is first installed), all files with the extension sv or SV are compiled as SystemVerilog files based on the settings in *formalpro.ini*. This behavior can be changed by changing the *formalpro.ini* settings or using one of the suffix control switches.

GUI Access

Location:	**Project** tab > **A** tab
	Project tab > **B** tab
Action:	Select **-sv** from the dropdown menu.

Examples

In the following example, *file1.v* on the A-side and *foo.v* on the B-side are treated as Verilog; *file2.sv*, *file3.sv*, and all similar files in *filelist* are treated as SystemVerilog 2009:

```
formalpro -a file1.v -sv file2.sv file3.sv -fl filelist -b foo.v
```

-sv2005

Scope: Design-specific

Alias: sv2005

Compiles files with SystemVerilog file extensions as SystemVerilog 2005 files.

Usage

- -sv *filename* [*filename*]...

  filename — Specifies a SystemVerilog file.

Description

This switch must precede design files to which it applies. This switch applies to all files within a design scope (-a, -b, or -common) including files specified in a file list or by any other means. See the examples below.

By default (when FormalPro is first installed), all files with the extension sv or SV are compiled as SystemVerilog files based on the settings in *formalpro.ini*. This behavior can be changed by changing the *formalpro.ini* settings or using one of the suffix control switches.

GUI Access

Location:	**Project** tab > **A** tab
	Project tab > **B** tab

Action: Select **-sv** from the dropdown menu.

Examples

In the following example, *file1.v* on the A-side and *foo.v* on the B-side are treated as Verilog; *file2.sv*, *file3.sv*, and all similar files in *filelist* are treated as SystemVerilog 2005:

```
formalpro -a file1.v -sv file2.sv file3.sv -fl filelist -b foo.v
```

-sv2009

Scope: Design-specific

Alias: none

Compiles files with SystemVerilog file extensions as SystemVerilog 2009 files.

Usage

- -sv2009 *filename* [*filename*]...

  filename — Specifies a SystemVerilog file.

Description

This switch must precede design files to which it applies. This switch applies to all files within a design scope (-a, -b, or -common) including files specified in a file list or by any other means. See the examples below.

By default (when FormalPro is first installed), all files with the extension sv or SV are compiled as SystemVerilog files based on the settings in *formalpro.ini*. This behavior can be changed by changing the *formalpro.ini* settings or using one of the suffix control switches.

GUI Access

Location: **Project** tab > **A** tab

Project tab > **B** tab

Action: Select **-sv2009** from the dropdown menu.

Examples

In the following example, *file1.v* on the A-side and *foo.v* on the B-side are treated as Verilog; *file2.sv*, *file3.sv*, and all Verilog files in *filelist* are treated as SystemVerilog 2009:

```
formalpro -a file1.v -sv2009 file2.sv file3.sv -fl filelist -b foo.v
```

-svFile

Scope: Design-specific

Alias: sv2009File

Compiles the specified file as SystemVerilog 2009.

Usage

- -svFile *filename*

  filename — Specifies a SystemVerilog file.

Description

Use this switch to compile the specified file as a SystemVerilog file, regardless of the file extension. The switch applies only to the specified file.

GUI Access

Location:	**Project** tab > **A** tab
	Project tab > **B** tab
Action:	Select **-svFile** from the dropdown menu to the left of the specific design file.

Examples

In the following example, all files are treated as Verilog except for *file2.v*, which is treated as SystemVerilog 2009.

```
formalpro -a file1.v -svfile file2.v file3.v -fl filelist -b foo.v
```

-sv2005File

Scope: Design-specific

Alias: sv2005File

Compiles the specified file as SystemVerilog 2005.

Usage

- -svFile *filename*

  filename — Specifies a SystemVerilog file.

Description

Use this switch to compile the specified file as a SystemVerilog file, regardless of the file extension. The switch applies only to the specified file.

GUI Access

Location:	**Project** tab > **A** tab
	Project tab > **B** tab
Action:	Select **-svFile** from the dropdown menu to the left of the specific design file.

Examples

In the following example, all files are treated as Verilog except for *file2.v*, which is treated as SystemVerilog 2005.

```
formalpro -a file1.v -svfile file2.v file3.v -fl filelist -b foo.v
```

-sv2009File

Scope: Design-specific

Alias: none

Compiles a specified file as a SystemVerilog 2009 file.

Usage

- -sv2009File *filename*

  filename — Specifies a SystemVerilog file.

Description

Use this switch to compile the specified file as a SystemVerilog file, regardless of the file extension. The switch applies only to the specified file.

GUI Access

Location:	**Project** tab > **A** tab
	Project tab > **B** tab
Action:	Select **-sv2009File** from the dropdown menu to the left of the specific design file.

Examples

In the following example, all files are treated as Verilog except for *file2.v*, which is treated as SystemVerilog 2009.

```
formalpro -a file1.v -sv2009file file2.v file3.v -fl filelist -b foo.v
```

-synopsysStrictArrayAddress

Scope: Design-specific

Alias: none

Enables array addressing compatible with Synopsys® Design Compiler® version 2006.06-SP1 and later.

Usage

- **-nosynopsysStrictArrayAddress** — handles address vectors consistent with pre-2006.06-SP1 versions of Design Compiler. Defaults to this behavior for Precision and Synplify flows using FVI and VIF.
- **-synopsysStrictArrayAddress** — handles address vectors consistent with 2006.06-SP1 and later versions of Design Compiler. Defaults to this behavior for Synopsys DC, Oasys RT, Cadence RTLC, Xilinx ISE, and Altera Quartus flows.

Description

Use this switch when comparing RTL to netlists that were compiled by Synopsys Design Compiler version 2006.06-SP1 and later. In pre-2006.06-SP1 versions, Design Compiler ignored the extra bits of an address vector that was larger than necessary for indexing the array, behavior consistent with the default behavior of FormalPro.

In current versions of Design Compiler, the extra bits are decoded (thus avoiding wrap-around read access). If this situation exists (and you have not specified this switch), FormalPro reports a difference for the associated target. The problem does not occur if the RTL code uses the correct-sized address vector.

Examples

```
formalpro -a -synopsysStrictArrayAddress file.v -b netlist.v -y library
```

GUI Access

Location: **Options** dialog box —

A specific pane > **RTL** tab

B specific pane > **RTL** tab

Action:	Enable: select **Synopsis strict array addressing** convention
	Disable: deselect **Synopsis strict array addressing** convention

-tlist

Scope: Global

Alias: None

The -tlist option enables the user to limit ECO analysis and patch generation to a subset of failing targets in the cache.

Usage

- -tlist *targetListFile* — Optional option specifying a file of targets that is used for ECO and difference region analysis.

Description

Optional option and string value that specifies a text file containing a list of targets to load for ECO analysis. The file specifies a list of target numbers as follows:

```
to9
tf27d
tf36q
```

To determine the targets for this file, you must run the FormalPro debugger on the complete set of targets and perform some initial analysis to determine which targets you want to load. Then create a *targetListFile* and restart the FormalPro tool loading the *targetListFile*.

GUI Access

Debugger tab > **Target list** entry box

Related Topics

-eco

-treatDivisionAsShift

Scope: Design-specific

Alias: None

Alters the behavior of FormalPro when compiling VHDL RTL code implementing the division of negative integers.

Usage

- -notreatDivisionAsShift — performs negative integer divisions according to the VHDL Language Reference Manual (default).
- -treatDivisionAsShift — performs negative integer divisions to mimic the behavior of Synopsys Design Compiler.

Description

When comparing a VHDL RTL design against a gate-level design synthesized with Synopsys Design Compiler, you may need to specify -treatDivisionAsShift to allow for a difference between how a simulator and Design Compiler handles negative integer division.

FormalPro issues a warning when compiling a negative-integer division and -treatDivisionAsShift is specified.

GUI Access

Location:	**Options** dialog box —
	A specific pane > **RTL** tab >
	B specific pane > **RTL** tab >
Action:	Enable: Select **Treat division as shift**
	Disable: Unselect **Treat division as shift**

Examples

```
formalpro -a -treatdivisionasshift ./rtl/design_a.vhd \
          -b ./gate/design_b.v
```

-upf

Scope: Design-specific

Alias: None

Specifies a UPF-compliant file containing power aware (PA) instructions for FormalPro compilation.

Usage

- -upf *upf_file*

  upf_file — Specifies a file containing UPF-compliant syntax.

Description

This option provides a way to pass UPF-compliant instructions to the FormalPro compilation. The compiler uses the information to generate the PA attributes required to produce a PA check report (see -PACheck).

A UPF-compliant file can be used in place of a legacy ModelSim PCF file.

UPF-compliant instructions can be applied to the compilation of RTL-level designs or gate-level designs. When providing UPF instructions for a gate-level compilation, be sure not to duplicate PA instructions that have already been implemented in the design by the synthesis tool that generated the netlist.

Examples

The -UPF option is design-specific and can be used more than once for each design side. For example:

```
-a -UPF file1.upf -UPF file2.upf
-b -upf test.upf
```

File1 and file2 are read in sequentially.

-useAliasPhases

Scope: Global

Alias: None

Performs complement matching when register aliases share the same name, but different phases.

Usage

- -useAliasPhases

Description

This switch instructs FormalPro to determine if a match can be made between alias names of different phases. If a match cannot be made, between the canonical names. This is illustrated in the following example, which can be seen in the Match Tool:

```
F \A.reg0
+ \A.abc.def

F \B.reg1
- \B.abc.def
```

In this case, FormalPro would not initially match \A.reg0 to \B.reg1 because their names are not the same, but upon analysis of their aliases, a complement match would occur because they have aliases with the same name, but only the phases (+ and -) are different.

GUI Access

Location:	**Options** dialog box —
	General pane > **Match** tab
Action:	Enable: Select **Use alias phases**
	Disable: Unselect **Use alias phases**

Examples

```
formalpro -usealiasphases -a design_a.v -b design_b.v
```

-V

Scope: Design-specific

Alias: None

Specifies a single technology library file.

Usage

- -v *file* — specifies the file location of a technology library. You can specify this switch any number of times on one command line.

Description

The library resolution scheme follows the behavior of Verilog-XL. For more information, see "Verilog Library Resolution" in the FormalPro User's Manual.

GUI Access

Location:	**Project** tab > **A** tab
	Project tab > **B** tab
	Project tab > **Common** tab
Action:	Type the path to your library or use: 
	Select **v** from the dropdown menu to the left of the library.

Examples

```
formalpro -a ./rtl/designa.v \
            -b ./gate/designb.v -v ./lib/library.v
```

-verifyTristate

Scope: Design-specific

Alias: None

Generates binary logic that models the Z-state of a tri-state buffer.

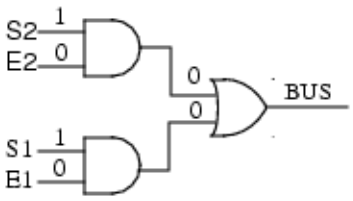
Usage

- -verifyTristate (default)
- -noverifyTristate

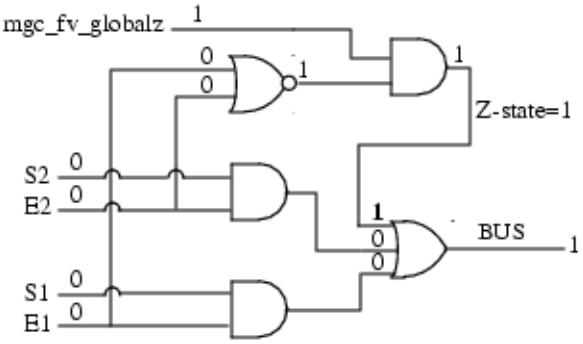
Description

When verifyTristate is enabled a special primary input port (mgc_fv_globalz) is generated to represent the Z-state in the logic. The globalz input participates in every instance where a tri-state buffer is modeled. To reduce clutter the globalz input is not displayed in reports and schematics. But if the globalz input feeds a target in which a difference is found, it is displayed in the debugger's schematics and reports so you can determine if a differing Z-state is the cause of the problem.

Figure 2-4. Two Modes of bufif Modeling



Tri-state verification disabled: bufifs are modeled as AND gates OR'd together; the Z-state is not represented in the



Tri-state verification enabled: a global "Z-state" input and additional logic models the Z-state.

Expect to see logic similar to this displayed in the debugger if a difference is found in target

-verilogFile

Scope: Design-specific

Alias: none

Compiles a specified file as Verilog

Usage

- -verilogFile *file*

  where *file* is a valid Verilog file.

Description

Use this switch to tell the FormalPro RTL compiler to treat the specified file as a Verilog file, regardless of the file extension. The switch applies only to the specified file.

GUI Access

Location: **Project** tab > **A** tab

Project tab > **B** tab

Action: Select **-verilogFile** from the dropdown menu to the left of the specific design file.

-version

Scope: Stand-alone

Alias: None

Displays FormalPro version information.

Usage

- -version

Description

Displays the version of the tool at the top of the FormalPro log files and in the lower right corner of the GUI.

GUI Access

Location: **Help** > **About** menu item

Examples

```
formalpro -version
```

-vcsCompat

Scope: Design-specific

Alias: none

Reduces FormalPro RTL compiler errors for VCS-compliant SystemVerilog files.

Usage

- -vcsCompat

Description

The FormalPro compiler adheres more stringently to the SystemVerilog LRM than the Synopsis VCS compiler, which may result in errors during compilation. By default, Precision returns

errors for any non-LRM compliant usages. To minimize incompatibilities, this switch affects the following constructs:

- Named port connection for bit/part select port is allowed. For example:

```
module top (i1, i2[2:0], i3, out1, out2);
input i1, i3;
input [2:0] i2;
output out1, out2;
bottom1 inst1 (.in1(i1), .in2(i2), .in3(i3), .o1(out1));
bottom2 inst2 (.in1(i1), .in2(i2[2]), .in3(i3), .o1(out2));
endmodule // top

module bottom1 (in1, in2[2:0], in3, o1, o2);
input in1,in3;
input [3:0] in2;
output o1,o2;
endmodule // bottom

module bottom2 (in1, in2[2], in3, o1, o2);
input in1,in3;
input [3:0] in2;
output o1,o2;
endmodule
```

- System task call inside a constant function call is allowed. For example:

```
module top;
function integer f(input integer x);
begin
$fflushall();
f = x;
end
endfunction
reg [f(1):1] r;
endmodule
```

- Macro name starting with a number is allowed. For example:

```
module test;
reg a;
`ifdef 11
reg b , c, d;
wire e, f;
`endif
assign e = (b + c + d);
endmodule
```

- Re-declaration of ANSI style port declaration with explicit data type in non-ANSI is allowed. For example:

```
module bug(
input clk,
input [31:0] in,
output [31:0] out
);
reg [31:0] out;
endmodule
```

- A function with no input port declaration. This LRM-compliant construct is not allowed. For example:

```
Module top;
function SIP;
endfunction
endmodule
```

- Reduction nand (~&) and reduction nor (~|) unary operators used as binary operators is allowed. For example:

```
out1 = a ~& b;
out2 = (rv_wr0 ~| rv_wr1);
```

- Verilog whole memory access in a system task/function is allowed. For example:

```
module top;
reg [31:0] data [0:1];
integer int, status;
initial begin
$spi4_dump ( int, data, status );
end
endmodule
```

- In a Verilog non-ANSI style port declaration, the explicit data type can be declared before its direction. For example:

```
module flop(ck,rst,din,si,se,so);
input ck,rst;
input si, se;
input din;
wire so; // (1) type declaration before
output so; //(1) port declaration later
endmodule
```

- Usage of an un-sized constant in concatenation is allowed with a warning. For example:

```
module top;
integer x;
initial begin
x = {1};
end
endmodule
```

- Constants connected on output and inout ports on the instance are allowed. For example:

```
module top(in1,in2,out);
input [7:0] in1,in2;
output reg [7:0]out;
middle m1 (in1,in2,2); //constant connected at output port
endmodule
module middle(in1,in2,out);
input [7:0] in1,in2;
output reg [7:0]out;
always @*
begin
out = in1;
end
endmodule
```

- 2-state data type for a net is allowed. For example:

```
module test15(in1,in2,out);
input [7:0] in1,in2;
output wire bit [7:0]out; //2 state data type bit used for net
declaration
assign out = in1;
endmodule
```

- Delay assignment in always_ff block is allowed. For example:

```
module top(clk,in1,in2,out);
input clk;
input [7:0] in1,in2;
output reg [7:0]out;
always_ff @(posedge clk)
begin
#2 out = in1;
end
endmodule
```

GUI Access

Location: **Options** dialog box —
A specific pane > **RTL** tab
B specific pane > **RTL** tab

Action: Enable: select **VCS compatibility**
Disable: deselect **VCS compatibility**

Examples

```
formalpro -mp 2 -a test.sv mydff.v -vcscompat -b netlist.v
```

-vhdl2008File

Scope: Design-specific

Alias: none

Compiles a specified file as VHDL 2008.

Usage

- -vhdl2008File *filename*

  where *filename* specifies the name of a VHDL file.

Description

This command compiles the specified file as VHDL 2008, regardless of the file extension.

GUI Access

Location:	**Project** tab > **A** tab
	Project tab > **B** tab
Action:	Select **-vhdl2008File** from the dropdown menu to the left of the specific design file.

Examples

In the following example design A consists of 2 design files compiled as VHDL 93 (default) format and design B has one file with a non-standard VHDL suffix compiled as VHDL 2008 format.

```
formalpro -a design_1a.vhd design_1b.vhd -b -vhdl2008File design_2.av
```

-vhdlFile

Scope: Design-specific

Alias: none

Compiles a specified file as VHDL.

Usage

- -vhdlFile *filename*

  where *filename specifies* the name of a valid VHDL file.

Description

This command compiles the specified file as VHDL, regardless of the file extension. The file is compiled in the VHDL format specfied with the -87 | -93 | -2008 switches. VHDL 93 is the default.

GUI Access

Location: **Project** tab > **A** tab

Project tab > **B** tab

Action: Select **-vhdlFile** from the dropdown menu to the left of the specific design file.

Examples

In the following example design A consists of 2 design files compiled as VHDL 2008 format and design B has one file with a non-standard VHDL suffix compiled as VHDL 93 format:

```
formalpro -a -2008 design_1a.vhd design_1b.vhd -b -93 -vhdlFile
design_2.av
```

-vlibF

Scope: Design-specific

Alias: None

Specifies a file listing Verilog technology library files.

Usage

- -vlibF *fileList* — specifies the location of a file list.

  fileList — the location of the file. One fileList argument per switch.

Description

You can specify this switch any number of times on the command line.

The format of the file specified by fileList is shown in the following example:

```
# commented line
./lib/ver_library_1.v            # a single Verilog library file
./lib/ver_library_*.v            # wildcards are allowed
```

GUI Access

Location:	**Project** tab > **A** tab **Project** tab > **B** tab **Project** tab > **Common** tab
Action:	Type the path to your filelist or use:  Select vlibF from the dropdown menu to the left of the library.

Examples

```
formalpro -a designA.v -b designB.v
   -common -vlibF ./verilog.fl
```

-vlog95 | -vlog01

Scope: Design-specific

Alias: None

Specifies that a design side (A or B) is written with either Verilog 95 or Verilog 2001 standards.

Usage

- -vlog95 — specifies Verilog 95.
- -vlog01 — specifies Verilog 2001 (default).

Description

You should specify **-vlog95** when you are using the FormalPro RTL compiler, your design does not use the Verilog 2001 standard, and your design uses Verilog 2001 keywords, such as **config**, **cell**, or **library**.

GUI Access

Location:	**Options** dialog box — **A specific** pane > **RTL** tab > **B specific** pane > **RTL** tab >
Action:	Enable: Select either **Verilog 95** or **Verilog 01**

Examples

The following example shows a case where your RTL design is not written with Verilog 2001 and you are using the RTL compiler.

```
formalpro   -a -vlog95 ./rtl/designA.v
            -b ./gate/designB.v
```

-vmapfile

Scope: Design-specific

Alias: None

Maps logical libraries within the design to a logical library of a different name.

Usage

- -vmapfile filename

  filename — contains the mapping information, and is formatted as follows:

  ```
  # commented line
  vmap <from_library> <to_library>
  ```

Description

Using this switch allows you to control VHDL logical library mapping with one switch, rather than having to use multiple -work switches on the command line.

GUI Access

Location: **Options** dialog box —
A specific pane > **RTL** tab >
B specific pane > **RTL** tab >
Vmap file entry box

Action: Type the path to your vmap file or use: 

Examples

Refer to the section “Specifying VHDL Library Files” in the FormalPro User’s Manual for an example on how to use this switch.

-work

Scope: Design-specific

Alias: -lib, -library

Maps VHDL, Verilog, and SystemVerilog files to a specified logical library.

Usage

- -work libraryName {[*fileType*] filePathname... | -fl fileList}

  libraryName — Specifies the name of a logical library. The default value is work.

  fileType — Optional switch that specifies the language type for the file using one of the following switches: -vhdl2008File *filename*, -verilogFile *filename*, or -sv2009 *filename*

  filePathname — repeatable argument that specifies the pathname of a file to map to the specified library.

  -fl fileList — maps all files listed in fileList to the libraryName.

Description

When your design or library files are dependent on logical libraries other than “work”, you must specify the dependencies on the command line using this switch.

GUI Access

Location:	**Project** tab > **A** tab **Project** tab > **B** tab **Project** tab > **Common** tab
Action:	1. Type the name of the library on a new line above the files associated with the library. 2. Select work from the dropdown menu to the left of the library name.

Examples

Example 1

The following example maps the *abc.vhd* and *def.vhd* files to a logical library named *shared* and maps the *top.vhd* file to a logical library named *work*:

```
formalpro -a -work share \
               ./rtl/abc.vhd \
               ./rtl/def.vhd \
               -work work \
               ./rtl/top.vhd \
          -b ./gate/design_b.v
```

Example 2

The following example maps a VHDL file with a non-standard file extension (*VhdlFile.vdl*) and *VhdlFile.vhd* to a logical library named *myworklib:*

```
formalpro -a -work myworklib
            -vhdlFile ../rtl/VhdlFile.vdl \
            ../rtl/myVhdlFile.vhd \
      -b ./gate/design_b.v
```

Example 3

The following example maps all the VHDL files listed in the *fileList* file to a logical library named *myworklib:*

```
formalpro -a -work myworklib
               -fl fileList \
          -b ./gate/design_b.v

fileList contents:
   ./rtl/abc.vhd
   ./rtl/def.vhd
   ./rtl/top.vhd
```

-y

Scope: Design-specific

Alias: None

Specifies directories containing technology library files.

Usage

- -y *directory* —specifies the directory location of technology libraries.

Description

The library resolution scheme follows the behavior of Verilog-XL. For more information, see “Verilog Library Resolution” in the FormalPro User’s Manual.

GUI Access

Location:	**Project** tab > **A** tab **Project** tab > **B** tab **Project** tab > **Common** tab
Action:	Type the path to your library directory or use:  Select **y** from the dropdown menu to the left of the library.

Examples

```
formalpro -a ./rtl/designA.v \
            -b ./gate/designB.v -y ./lib/library/
```

-ylibF

Scope: Design-specific

Alias: None

Specifies a file that lists one or more directories containing Verilog technology library files.

Usage

- -ylibF *dirList* — Species a file containing a list of one or more directories. One dirList file per switch.

Description

This option provides a way to specify one or more directories that contain Verilog library files to use for design compilation. *dirList* itself is a file that contains a list of one or more directories in which the library files reside.

You can -ylibF multiple times on the command line.

The format of the file specified by dirList is shown in the following example:

```
# commented line
./lib/ver_library_1              # a directory of Verilog libraries
./lib/ver_lib_*                  # wildcards are allowed
```

GUI Access

Location:	**Project** tab > **A** tab **Project** tab > **B** tab **Project** tab > **Common** tab
Action:	Type the path to your filelist or use:  Select **ylibF** from the dropdown menu to the left of the library.

Examples

```
formalpro -a designA.v -b designB.v \
            -common -ylibF ./list_of_libs.ylibs
```

Chapter 3
FPGA Tools

FormalPro provides an FPGA-only version of the formalpro command: *formalpro_fpga***.** *formalpro_fpga* enables you to run an FPGA flow with reduced cost licenses including two utilities that generate the set-up files required by FormalPro for post-synthesis equivalence checking.

Table 3-1. FPGA Tools and Licenses

Command	Description/License
formalpro_fpga	FormalPro (GUI and command-line functionality) for FPGA designs. Required license: formalpro_fpga or formalpro.
transFVI	Utility the automates post-synthesis equivalency checking by translating a Precision FVI file into the FormalPro set-up files. Required license: any.
transVIF	Utility that automates post-synthesis equivalency checking by translating a SynplifyPro VIF file into FormalPro set-up files. Required license: any.

formalpro_fpga

Invokes FormalPro for FPGA designs only. All FormalPro GUI and all command-line switches are available.

Usage

formalpro_fpga -fpga {altera | actel | xilinx} {-fvi file.fvi | -wsp *file*.wsp | *formalpro_options*}

Arguments

- -fpga {altera | actel | xilinx}

  Specifies the FPGA vendor. If excluded or if the specified vendor does not match the actual vendor, FormalPro exits. Can be omitted when starting the FormalPro GUI (-gui), but the vendor must be specified from the GUI before starting the verification run.

- -fvi *file*.fvi

  Runs transFVI on the specified fvi file, then starts the formalpro run using the command-line options in the WSP file generated by transFVI.

- -wsp *file*.wsp

  Runs FormalPro using the command-line options in the specified workspace file (.wsp) generated by either transFVI or transVIF.

- *formalpro_options*

  One or more formalpro command-line option (for a complete list see the formalpro Command command page). If you use the -fvi or -wsp option, the options you specify here are added to the options specified in the WSP file.

Examples

This example runs post-synthesis (Precision) equivalence check on an FPGA design that uses Altera components. The synthesis process has produced an FVI file. At the start of the FormalPro run, the FVI file is processed by transFVI, producing a WSP file that contains the command line options for the FormalPro run:

```
formalpro_fpga -fpga altera -fvi infile.fvi
```

(Note that the *.fvi* file contains the vendor flag, so "-fpga altera" is not required here.)

In this example a previously generated WSP file (*infile.wsp*) is specified:

```
formalpro_fpga -fpga altera -wsp infile.wsp
```

This example starts a GUI session using a formalpro_fpga license:

```
formalpro_fpga -gui -fpga xilinx
```

transFVI

Translates an FVI file into the FormalPro files required for post-synthesis equivalence checking.

Usage

transFVI [-vendor {actel | altera | xilinx}] [-y technology_library]
[-path_map string=*newstring*]... infile.fvi

FormalPro command-line invocation use:
-fvi *infile*.fvi

Arguments

- -v[ender] {*actel* | *altera* | *xilinx*}

  This specifies the vendor used in the design source. The vendor is typically included in the generated FVI file. If it is missing or incorrect, transFVI will produce faulty output files. Explicitly specifying the vendor on the transFVI command line ensures that the vendor specification in the generated FormalPro files is correct. This switch overrides existing information in the FVI file.

- -y directory_path

  This specifies the path to the FPGA verification library. You may include this option on the command line multiple times, once for each verification library.

- -path_map string=newstring ...

  The files generated by transFVI contain strings that specify the full path to the design source, extracted from the FVI file generated by Precision RTL. The source paths that were valid at Precision RTL runtime may not be valid at FormalPro runtime.

  The -path_map option uses a simple string replacement to transform paths, replacing *string* with *newstring*, where ever *string* occurs. This option can be repeated on the command line. You cannot use this option if the environment variable FORMAL_PATH_MAP is set. For more information, see the Environment Variables section at the end of this section.

  Example1: At Precision runtime the source files are in /home/design/myfiles. At FormalPro runtime they will be in /home/verification/myfiles:

  ```
  -path_map /home/design=/home/verification
  ```

  Example2: The Precision runtime environment is Windows. The FormalPro runtime environment is UNIX. On the transFVI commandline backslashes must be escaped (\\) or the entire map string must be enclosed in quotation marks:

  ```
  -path_map c:\\path\\design=/home/verification
  ```

  or

  ```
  -path_map c:"\path\design=/home/verification"
  ```

Example 3: You can specify multiple mappings by repeating the -path_map option on the command line. The transformations are applied in the order they appear on the command line:

```
transFVI -precision \
-path_map "c:\VHDL\path1\=/user/ausome/VHDL/"\
-path_map "c:\VHDL\path2\=/user/ausome/VHDL/"\
-path_map "c:\VHDL\path3\=/user/ausome/VHDL/"
```

Description

The transFVI utility takes as input *infile*.fvi (an FVI file generated by Precision RTL) and outputs the set-up files used by FormalPro for a post-synthesis equivalence check. transFVI can be invoked as a separate utility (see first usage), or can be invoked as an command-line option in the FormalPro command-line invocation (see -fvi on the formalpro_fpga page).

The setup files generated by transFVI are:

infile.bb — blackbox file

infile.config — configuration file

infile.constr — constraint file

infile.flow — flow file

infile.match — explicit-match file

infile.rule — match-rule file

infile.wsp — workspace file containing the FormalPro command line

Environment Variables

FORMAL_PATH_MAP

The environment variable FORMAL_PATH_MAP provides the same functionality as the -path_map option. If FORMAL_PATH_MAP is set, you cannot use the -path_map option.

The string mapping must be enclosed in quotes. You can specify multiple string mappings, separated by a space.

For example:

```
setenv FORMAL_PATH_MAP \
"c:\VHDL\path1\=/user/ausome/VHDL/ \VHDL\path2\=/user/ausome/VHDL/"
```

transVIF

Translates a VIF file generated by SynplifyPro into the FormalPro setup files required for post-synthesis equivalency checking.

Usage

transVIF [-y technology_library] infile.vif

Arguments

- -y directory_path

  This specifies the path to one, and only one, FPGA verification library. You may include this option on the command line multiple times, once for each required library.

Description

The transVIF utility takes as input a VIF file generated by SynplifyPro and outputs the FormalPro files required for a post-synthesis equivalency check. One of output files is a workspace file (.wsp) containing the complete command line invocation for starting the equivalency check.

The transVIF utility expects an input file with a .vif extension. Assuming an input file named **infile.vif,** transVIF outputs these FormalPro files:

infile.bbox — blackbox file

infile.conf — configuration file

infile.constraint — constraint file

infile.flow — flow file

infile.user — explicit-match file

infile.rule — match-rule file

infile.wsp — workspace file containing the FormalPro command line

Chapter 4
Debugger Commands

The *fpdebug* command invokes the FormalPro debugger. Once invoked, you can use the debugger shell commands to debug design differences.

The debugger shell commands are described in Table 4-1.

For more information on using the FormalPro debugger, see Debugging Design Differences in the *FormalPro User's Manual*.

fpdebug Command

Invokes the FormalPro debugger program shell.

Usage

fpdebug [-cache *cacheDir*] [*diffNetworkName*] [-f *command_file*] [-tlist *targetListFile*] [-tmax *number*] [-jnl *journalFile*] [*targetTypes*]

Description

When invoked, the FormalPro debugger program extracts a Verilog *difference network* and provides access to set of interactive commands you can use in the debugger shell to assist in finding the location of design differences. A *difference network* is a logic network of failing targets.

If the FormalPro comparison ended with a high number of different targets, you can debug a subset of these targets by using the **-tlist** or **-tmax** switches. If you choose to use either of these switches, the difference network is saved in Verilog format in the directory formalpro.cache/debug.

After extracting the difference network, the debugger performs *auto network learning*, which is a form of structural analysis, on the network to merge similar gates between the A and B networks. This simplifies the difference network.

Statistical information (numbers of inputs, outputs, and gates) displays while the difference network is extracted. Based on this data, the debugger recommends a set of instructions for the interactive debug process.

The transcript information is save to the file *fpdebug.log* in the *logs* directory of the FormalPro cache. All the debug commands entered are also saved to the journal file *fpdebug.jnl* in the *debug* directory of the FormalPro cache.

Arguments

- -cache cacheDir

  Optional switch and string value that specifies a FormalPro cache directory to extract the difference network from for the debug session. By default, the difference network is extracted from and saved to the FormalPro cache in the current directory. The difference network file is saved in a debug directory and named *difference_1.v* file in the FormalPro cache.

  GUI Location

  Project tab >**General** tab > **Cache directory** entry box

- diffNetworkName

  Optional string value that specifies a difference network file to load for the debug session. By default, the difference network from the current directory is loaded: formalpro.cache/debug/difference_1.v

  GUI Location

  Debugger tab > **Difference net** entry box

- -f *commandFile*

  Optional switch and string value that specifies a text file to load for the debug session. This text file can contain debug commands to execute automatically or be any other text file created by the FormalPro debugger you want loaded. The format of the commandFile used for debug commands is as follows:

  ```
  # commented line
  <command> ... \
  <command> ...
  ```

  Comment out all newline characters and use a back-slash (\) when specifying options across multiple lines.

  GUI Location

  Debugger tab > **Command file** entry box

- -tlist targetListFile

  Optional switch and string value that specifies a text file containing a list of targets to load for the debug session. The file specifies a list of target numbers as follows:

  ```
  to9
  tf27d
  tf36q
  ```

  To determine the targets for this file, you must run the FormalPro debugger on the complete set of targets and perform some initial analysis to determine which targets you want to load, create a targetListFile, and restart the debugger loading the *targetListFile*.

  GUI Location

  Debugger tab > **Target list** entry box

- -tmax number

  Optional switch and integer value that specifies the maximum number of random targets to load for the debug session.

  GUI Location

Debugger tab

- -jnl journalFile

  Optional switch and string value that specifies a name and location for the journal file. The journal file records all debug commands issued during a debug session. By default, the journal file is saved in the following location:

  ```
  ./formalpro.cache/debug/fpdebug.jnl
  ```

  GUI Location

  Debugger tab > **Journal file** entry box

- *targetTypes*

  Optional series of switches that specify the types of targets to load for the debug session. Switch options include:

  -cycles — loads unsolved targets due to being *fed by cycles*.

  -diffs — loads targets designated as *not equivalent* (default).

  -floats — loads unsolved targets due to being *fed by floating net*.

  -md — loads unsolved targets due to being *fed by multiply driven nets*.

  -unsolved — loads targets designated as *unsolved*.

  -unmatched — loads targets designated as *fed by unmatched*.

  You can specify any of these switches on the command line, in any order. Note that if you specify any switches, the default switch (-diffs) is not used unless you specify it.

  GUI Location

  Debugger tab

- -help

  Optional switch that displays help for the *fpdebug* command and a list of the debugger shell commands.

Examples

The following example shows how to load a custom-named difference network from a non-standard FormalPro cache directory.

```
fpdebug -cache formalpro_new.cache difference_new.v
```

The following example loads all targets listed in targets.list and creates a difference vector, difference_user.v, based on those targets.

```
fpdebug -tlist targets.list difference_user.v
```

The following example loads all targets designated as *fed by cycle* or *fed by floating net* from the FormalPro cache named alternate.cache.

```
fpdebug -cache alternate.cache -cycles -floats
```

Debugger Shell Commands

The *fpdebug* command invokes the FormalPro debugger. Once invoked, you can use the debugger shell commands to debug design differences.

The debugger shell commands are described in Table 4-1.

For more information on using the FormalPro debugger, see Debugging Design Differences in the *FormalPro User's Manual*

Table 4-1. Debugger Command Summary

Command	Description
Simplification Commands	
networklearn	Determines if the identified gate pairs are functionally equivalent. If the gate pairs are found to be equivalent, the design B subcone is merged into design A.
checkequiv	Determines if two gates, one from design A and the other from design B, are functionally equivalent.
addtarget	Creates a new target based on two comparison points (one from each design); once the new target is created, you can run an analysis on it.
pairgates	Identifies gates that should have functionally equivalent behavior, which the debugger has not already detected.
Investigation Commands	
analyze	Investigates the logic driving each *target* and the logic structure and functionality of both the A and B networks. This command also prunes the error region contained in B, given that A is the reference.
drives	Provides information about the number of targets driven by the netName or gateID.
btc	Checks specified targets for buffer tree parity.
statistics	Displays statistics about the logic fan-in structure driving the selected *target*(s).
nodeinfo	Displays information about the specified schematic instance or gate name.
whatif	Internally modifies the functionality of the difference network, based on the *whatifArgument*, then queries the difference network to determine the equivalency of the affected target(s).
extracteco	Main entry command for ECO and difference region analysis capabilities. Analyze difference regions and extract or generate patch information for ECO and debug use models.
Schematic Display Command:	

Table 4-1. Debugger Command Summary (cont.)

Command	Description
showschematic	Examines the difference network logic structure as it exists at invocation and displays a schematic representation of the network.
Report Commands	
eqnetreport	Generates a report identifying functionally equivalent nets between the two designs.
pinpointreport	Generates a report identifying potential areas of error discovered by the debugger.
Save Commands:	
savenetwork	Writes out the logic difference network as it exists at the point of invocation.
extracttarget	Writes to a file a logic difference network for selected targets.
Miscellaneous Commands	
help	Displays help the **fpdebug** command and debugger shell commands.
syntax	Displays the syntax for the debugger commands.
quit	Exits the debugger shell without saving.

addtarget

Creates a new target based on two comparison points (one from each design).

Usage

{addtarget | add} ***gateA gateB***

Arguments

- ***gateA***

  An argument that specifies a comparison point from design A.

- ***gateB***

  An argument that specifies a comparison point from design B.

Description

Once the new target is created, you can run an analysis on it.

This command is useful when you think that comparison points should have similar behavior, but they display different logic values in a schematic display (**showschematic -diff**). By using **addtarget** on these two gates, and then performing an analysis on the newly created target (**analyze**), you can quickly identify the state and primary input assignments that forced these two gates to diverge.

GUI Access

From a Target Schematic or Target Difference Schematic that shows the comparison points you want to use to create a new target:

1. Select the A side comparison point.
2. Select the B side comparison point.
3. Click and hold the **Gate Pair Functions** button. 
4. From the exposed drop-down menu, select **Add target**.

   The Debug tab is displayed and the addtarget command is issued.

```
debug> addtarget U188 U263
..... addtarget \A.uart_top.U404.Y \B.uart_top.U404.Y
..... Added new target to900001
```

You can now perform various debug tasks on this new target, such as Statistics or Analyze.

Examples

The following command creates a target between the two gates: \A.pgmbl_m8051.m8051.U10.i_2335.Z and \B.pgmbl_m8051.m8051.U10.i_2335.Z.

```
fpdebug> add \A.pgmbl_m8051.m8051.U10.i_2335.Z
\                    \B.pgmbl_m8051.m8051.U10.i_2335.Z
```

The following command creates a target between the two gates U1152 and U1160. This example uses internally generated gate names, as shown in the schematic viewer of the FormalPro GUI.

```
fpdebug> add U1152
U1160
```

analyze

Investigates the driving logic and logic structure and functionality of both the A and B networks.

Usage

{analyze | ana} [-bdd] [-atpg] [-rps] [-engine] [*target* …]

Arguments

- -bdd

  An argument that uses a difference vector discovery algorithm based on binary decision diagrams.

- -atpg

  An argument that uses a difference vector discovery algorithm based on ATPG.

- -rps

  An argument that uses a difference vector discovery algorithm based on random pattern simulations.

- -engine

  An argument that uses a difference vector discovery algorithm based on the difference vector discovered during the Solve stage of FormalPro.

- *target* …

  An argument that specifies the target to be analyzed. You can specify this argument any number of times from the command prompt. The default behavior is for the **analyze** command to perform analysis on every target in the difference network.

Description

Investigates the logic driving each *target* and the logic structure and functionality of both the A and B networks. This command also prunes the error region contained in B, given that A is the reference.+

During this analysis, the debugger records difference vectors. In addition, the difference vector that has the smallest number of necessary primary input assignments is stored for later use by the **showschematic -diff** command.

If you do not provide a list of *target* arguments, the debugger performs analysis iteratively on every target in the difference network.

The **analyze** command also uses the **networklearn** command on individual targets.

You should use this command without specifying a *target* only when the number of targets in the difference network is small. Otherwise, first use the **statistics** command to choose targets for analysis.

At the completion of the **analyze** command, the debugger might have discovered some additional candidate gates that could be checked for equivalency. The debugger stores these candidate gate pairs for later use with the **checkequiv** command to check their equivalency.

By default, analyze uses the four different search methods (-bdd, -atpg, -rps, and -engine) to discover a difference vector. You can specify which methods are to be used by including the method switches on the command line.

GUI Access

From any debug transcript or report that shows the target you wish to analyze:

1. Right-click the target name, such as **tf19d**.
2. From the exposed drop-down menu select the option **Analyze**.

   The Debug tab is displayed and the analyze command is issued.

Examples

```
fpdebug> ana to90

fpdebug> ana to98
to99
```

btc

Checks specified targets for buffer tree parity.

Usage

btc ***target...***

Arguments

- ***target...***

  Space separated list of target names to check for buffer tree parity errors.

Description

Checking for buffer tree parity is useful when comparing gate-to-gate designs that have a high degree of structural similarity.

When formalpro matches up input ports and registers, it uses those points to create a series of pseudo primary inputs for each matched input port and matched register (in addition to pseudo primary outputs for each register). These pseudo primary inputs are used to tie the matched logic nets or matched ports from design A and B for equivalency checking.

The buffer tree parity check examines the parity of the inverter/buffer chains driven by the pseudo primary input into both design A and design B up to either the first multiple input gate or a net with multiple fanouts. When a parity difference is detected during the check, a Buffer Tree Parity Check (.btc) file is created. For each parity check error found, a report file is created at *formalpro.cache/debug/<target>.btc*.

The .btc file lists the failing target name and associated target comparison points along with a trace of the nets that make up the checked parity chain.

The following example shows the contents of a .btc file:

```
# Target to2 buffer tree parity check
#  Design A comparison pt: \A.top.rdw_status
#  Design B comparison pt: \B.top.rdw_status
    Buffer/Inverter chain paths reach gates with different chain parities
(i0).
    A design chain:
     + \A.top.read
     + \A.top.read32
     + \A.top.read32_dly
    B design chain:
     + \B.top.read
     + \B.top.read32
     - \B.top.read_32n
```

The first entry in chains is the port/register name. The +/- tracks parity of chain (- inverted).

Though not shown above, if a complimentary match was made for the 1st chain entry, the parity setting would show asterisk (*) after the name with the message "* Note: defined as complementary match point."

When parity check errors occur, you should check the logic for a design error, a potential incorrect library model, or an error in FormalPro determination of complementary matching.

Examples

The following command checks for buffer tree parity on the following targets: to1, tf11q, tl, and 100q:

```
fpdebug> btc to1 tf11q
tl100q
```

checkequiv

Merges gate from design A to verify equivalence.

Usage

{checkequiv | chk} [*gateA gateB*]

Arguments

- *gateA*

  An argument that specifies a gate from design A.

- *gateB*

  An argument that specifies a gate from design B.

Description

Determines if two gates, one from design A and the other from design B, are functionally equivalent.

If the two gates are equivalent, the debugger automatically merges the gate in design A to drive the fanout logic in design B. This assumes that if the gates are functionally equivalent, none of the logic nets that fan into gate B can be the source of error.

If you do not provide any arguments, **checkequiv** runs iteratively on the entire list of potential functionally-equivalent gate pairs stored during target and/or network analysis.

Normally, the **networklearn** command finds many of the functionally equivalent gates. However, network learning might not have solved the equivalency verification problem because it has a short run time before time-out. **checkequiv**, with its longer run time before time-out, might be more successful.

GUI Access

From a Target Schematic or Target Difference Schematic that shows the comparison points of which you want to check equivalence:

1. Select the A side comparison point.
2. Select the B side comparison point.
3. Click and hold the **Gate Pair Functions** button. 
4. From the exposed drop-down menu, select **Check equivalent**.

   The Debug tab is displayed and the addtarget command is issued.

```
debug> checkequiv U188 U263
..... checkequiv U188 U263
       NOT EQUIVALENT!
```

Examples

The following command verifies the equivalence between the two gates U1152 in design A and U1160 in design B. This check uses the internally generated names, as shown in the schematic viewer of the FormalPro GUI.

fpdebug> chk U1152
U1160

drives

Lists the targets driven by a specified net or target.

Usage

drives [-l] {***netName*** | ***gateID***}

Arguments

- -l

  An argument that instructs the debugger to list the targets driven by the netName or gateID in addition to the target ratio information.

- ***netName***

  An argument that specifies the name of a net within the difference network.

- ***gateID***

  An argument that specifies the ID number of a gate in the difference network.

Description

Provides information about the number of targets driven by the netName or gateID.

The information includes a ratio of the targets driven to the total number of targets.

GUI Access

There is no specific GUI function to perform this task. However, the Nodeinfo command does produce information equivalent to the **drives -l** command in the **Targets driven** section of the report.

From any debug transcript or report that shows the net you wish to analyze:

1. Right click on an object name, such as **A.uart_top.I4/xmitdt_reg[6].SFFR.Q_ppo**.
2. From the exposed drop-down menu select the option **Nodeinfo**.

   The Debug tab is displayed and the nodeinfo command is issued.

```
debug> nodeinfo {\A.uart_top.I4/xmitdt_reg[6].SFFR.Q_ppo}

... nodeinfo \A.uart_top.I4/xmitdt_reg[6].SFFR.Q_ppo
  Status: U1355 PAIRED
  Orig Netlist:
    buf ( \A.uart_top.I4/xmitdt_reg[6].SFFR.Q_ppo ,
                                              \A.uart_top.I4/
xmitdt_reg[6].SFFR.Y );
  Fansout To:
      tf19d
      mf19t2
  Paired Mate: U1376
  Targets driven:
    tf19d  (\A.uart_top.I4/xmitdt_reg[6].SFFR.Q_ppo)
    tf19q  (\A.uart_top.I4/xmitdt_reg[6].SFFR.Q_ppo)
..... 2/37 (5.41%) targets are driven by \A.uart_top.I4/
xmitdt_reg[6].SFFR.Q_ppo
```

Examples

The following command returns information on targets in the fan out of gate U1314:

fpdebug> drives U1314

The following command returns detailed information on gate \B.m8051.U10.Z:

fpdebug> drives -l
\B.m8051.U10.Z

eqnetreport

Generates a report, *fileName*, identifying functionally equivalent nets between the two designs.

Usage

{eqnetreport | enr} ***fileName***

Arguments

- ***fileName***

  An argument that specifies the location and name of the report the debugger generates. By default, this file is saved in the *debug* directory of the FormalPro cache.

Examples

```
fpdebug> enr equiv_net.report
```

extracteco

Depending on switches used, this command analyzes difference regions and extracts or generates patch information for ECO and debug use models.

Usage

extracteco [-generate] [-final *directoryPath*] [-correspond *fullNameA fullNameB*]

Arguments

- -generate

  Analyzes the current miter and failing compare points and generates patch data in the ECO patch DB in the *<cache>/eco* directory. The -generate switch is mutually exclusive with the -final switch.

- -final *directoryPath*

  The -final switch is used to "finalize" ECO patch data as described earlier. The *directoryPath* field is optional and is used to specify a directory where the output patch files are written. By default, patch data is written into the *<cache>/eco* directory. The -generate switch is mutually exclusive with the -final switch.

- -correspond *fullNameA fullNameB*

  The user in some cases can help the patch quality by providing functional correspondence pairs to the tool. This switch provides an interactive way the user can identify a pair of signals (one on the A side, one on the B side) as a functional correspondence point.

Description

The extracteco command is expected to be called after an ECO design and original layout netlist have been compared. If the user wishes to generate a patch for a subset of the failing points, then the -tlist switch is used with formalpro command as described earlier. It is expected that the user will not use the command very often; it is provided for GUI access needs.

GUI Access

Location:	Tools tab > **ECO Operations** > **Extract ECO**
Action:	Same as extracteco -generate.
Location:	Tools tab > **ECO Operations** > **Finalize ECO**
Action:	Same as extracteco -final.
Location:	Tools tab > **ECO Operations** > **Add Correspondence Constraint**
Action:	Same as extracteco -correspond.
Location:	Debug Tab > **Difference Region Analysis**
Action:	Same as extracteco -generate.

Related Topics

-eco

-tlist

eco_correspond

extracttarget

Writes out a logic difference network for selected *target*(s) to the file *fileName*.

Usage

{extracttarget | ext} [-f *fileName*] {***target*** …}

Arguments

- -f *fileName*

  A switch and argument pair that specifies the name of the extracted difference network. If you do not specify this switch, the default *fileName* is Target_*targetName*.v, where *targetName* is the specified **target**, and is saved in the *debug* directory of the FormalPro Cache. If you specify more than one **target**, the default *fileName* is FDDT_Targets.v.

- ***target*** …

  An argument that specifies the target to be extracted. You can specify this argument any number of times.

Description

Use this command when the difference network is large and has many failing targets. It works on a subset of the difference network, which is usually more efficient. Re-invoke the debugger with the subset file *fileName*. For example, if all bits of a bus are failing targets, it is likely that there is a single source of difference. In that case, working on a single bit is more productive.

Examples

fpdebug> ext to98

fpdebug> ext -f diffnet_t099.v

help

Displays help information for the **fpdebug** command and debugger shell commands.

Usage

help

Arguments

None

Examples

fpdebug> help

networklearn

Finds logic gate pairs that are potentially equivalent.

Usage

{networklearn | learn} [-l | -m | -h]

Arguments

- -l

  A switch that performs an analysis requiring a low amount of CPU time. This is the default behavior if you do not specify a switch to this command.

- -m

  A switch that performs an analysis requiring a medium amount of CPU time.

- -h

  A switch that performs an analysis requiring a high amount of CPU time.

Description

Examines the network structure and finds logic gate pairs that are potentially equivalent.

Note

The debugger performs this command automatically on individual targets when you run the **analyze** command.

This command determines if the identified gate pairs are functionally equivalent. If the gate pairs are found to be equivalent, the design B subcone is merged into design A.

Although the equivalency checking algorithm is complete, it might not be able to determine equivalency because of time constraints. The options **-l**, **-m**, and **-h** control the efficiency of the gate learning by allocating varying CPU-time analysis. If you are not extracting a subset of the difference network for use in another invocation of **fpdebug**, you should enter **networklearn** first.

Examples

```
fpdebug> learn -m
```

nodeinfo

Displays information about the specified schematic instance or gate.

Usage

{nodeinfo | ni} {***netName*** | ***gateID***}

Arguments

- ***netName***

  An argument that specifies the name of a net within the difference network.

- ***gateID***

  An argument that specifies the ID number of a gate in the difference network.

Description

The information includes the full name of the net, its pairing status, which gates it drives, and targets in its fanout.

GUI Access

From any debug transcript or report that shows the net to analyze:

1. Right click on an object name, such as **\A.uart_top.I4/xmitdt_reg[6].SFFR.Q_ppo**.
2. From the exposed drop-down menu select the option **Nodeinfo**.

   The Debug tab is displayed and the nodeinfo command is issued.

```
debug> nodeinfo {\A.uart_top.I4/xmitdt_reg[6].SFFR.Q_ppo}

... nodeinfo \A.uart_top.I4/xmitdt_reg[6].SFFR.Q_ppo
  Status: U1355 PAIRED
  Orig Netlist:
    buf ( \A.uart_top.I4/xmitdt_reg[6].SFFR.Q_ppo ,
                                                \A.uart_top.I4/
xmitdt_reg[6].SFFR.Y );
  Fansout To:
      tf19d
      mf19t2
  Paired Mate: U1376
  Targets driven:
    tf19d  (\A.uart_top.I4/xmitdt_reg[6].SFFR.Q_ppo)
    tf19q  (\A.uart_top.I4/xmitdt_reg[6].SFFR.Q_ppo)
..... 2/37 (5.41%) targets are driven by \A.uart_top.I4/
xmitdt_reg[6].SFFR.Q_ppo
```

Examples

The following command returns information on target *to99*:

fpdebug> ni to99

The following command returns information on gate \A.pgm.m8051.U10.i_23.Z:

```
fpdebug> ni \A.pgm.m8051.U10.i_23.Z
```

The following command returns information on a gate ID (U1160), as shown in the schematic viewer of the FormalPro GUI:

```
fpdebug> ni U1160
```

pairgates

Identifies gates with equivalent behavior.

Usage

{pairgates | pair} ***gateA gateB***

Arguments

- ***gateA***

  Specifies a gate from design A.

- ***gateB***

  Specifies a gate from design B.

Description

Identifies gates that should have functionally equivalent behavior, which the debugger has not already detected.

GUI Access

From a Target Schematic or Target Difference Schematic that shows the comparison points of which you want to identified as paired:

1. Select the A side comparison point.
2. Select the B side comparison point.
3. Click and hold the **Gate Pair Functions** button. 
4. From the exposed drop-down menu, select **Pair gates**.

   The Debug tab is displayed and the addtarget command is issued, using the internal identifiers.

   ```
   debug> pairgates U188 U263
   ..... pairgates U188 U263
   ```

Examples

```
fpdebug> pair \A.pgmbl_m8051.m8051.U10.i_2335.Z
\                \B.pgmbl_m8051.m8051.U10.i_2335.Z
```

pinpointreport

Identifies potential areas of error discovered by the debugger.

Usage

{pinpointreport | ppr} [-win] ***fileName***

Arguments

- -win (GUI mode only)

  A switch that displays the contents of the report in a text window in addition to writing the information to *fileName*.

- ***fileName***

  An argument that specifies the location and name of the report the debugger generates. By default, this file is saved in the debug directory of the FormalPro cache.

Description

Use this command after performing analyze on the desired targets.

The debugger reports the percentage of comparison points that you can be obtain from the error site, and the report is sorted by the percentage figure.

This report is generally useful on designs similar in structure. The percentage of targets solved by the isomorphism engine is a good indicator of structural similarity.

Examples

```
fpdebug> ppr ./fpdebug_reports/pinpoint.report
```

quit

Exits the debugger shell without saving.

Usage

quit

Arguments

None

Examples

fpdebug> quit

savenetwork

Writes out the logic difference network as it exists at the point of invocation.

Usage

{savenetwork | save} ***fileName***

Arguments

- ***fileName***

  An argument that specifies a file name.

Examples

```
fpdebug> save diffnet_test.v
```

showschematic

Displays a schematic representation of the difference logic structure (only available from the GUI).

Usage

{showschematic | show} [-diff] [-merge] [*target*]

Arguments

- -diff

  A switch that displays a schematic showing the effect of simulating a difference vector on the target. The schematics display values on the output of every gate with a logic 0 or logic 1. The schematics only display gates that contribute to the difference or its propagation.

- -merge

  A switch that displays a single schematic containing designs A and B with all merged gates displayed only once. If you specify *target*, the schematic is limited to the cone of logic in the fan-in of this target; this is the recommended syntax.

- *target*

  An optional argument that specifies the target(s) to be extracted.

Description

Examines the difference network logic structure as it exists at invocation and displays a schematic representation of the network.

The -MERGE switch displays one schematic for design A and B, with all merged gates displayed only once.

Within the schematics, the gates will be color-coded as follows:

Yellow	Comparison point (merged schematic).
Red	Gates in both designs that are potential sources of error.
Green	Paired gate in design A to a gate in design B.
Blue	Paired gate in design B to a gate in design A.
Gray	Gates that are structurally equivalent between the designs.
Purple	Nets that are required to exhibit the difference.

You can specify the color for the schematic from **Edit > Preferences**. For more information, see “Main Menu Bar” in the *FormalPro User’s Manual*.

Examples

```
fpdebug> show -merge
-diff to99
```

**fpdebug> show -diff
to99**

statistics

Displays statistics about the logic fan-in structure driving the selected *target*(s).

Usage

{statistics | stats} [-file] {***target*** …}

Arguments

- -file

  An argument that allows you to write the results of this command to the *target_stats.txt* file in the *debug* directory of the FormalPro cache. By default, this information is written to stdout and the Debug log.

- ***target*** …

  An argument that specifies the target(s) to display statistics about. You can specify this argument any number of times.

Description

If you do not specify a *target*, the debugger displays statistics for all failing targets.

The statistics include instance count, the number of equivalent gates and paired gates, and the combinational depth.

GUI Access

From any debug transcript or report that shows the target you wish to analyze:

1. Right-click the target name, such as **tf19d**.
2. From the exposed drop-down menu select the option **Stats**.

   The target_stats tab is displayed containing information about the target.

Examples

```
fpdebug> stats to99
```

```
fpdebug> stats to99
to98
```

syntax

Displays the syntax for the debugger commands.

Usage

{syntax | syn}

Arguments

None

Examples

fpdebug> syn

tdvr

Analyzes the difference vectors for each target .

Usage

tdvr [-f *fileName*] [-win]

Arguments

- -f *fileName*

  A switch and argument pair that specifies the name to save the information as. This information is saved in the debug directory of the FormalPro cache. If you do not specify this switch, the information displays in the fpdebug.log as well as stdout.

- -win (GUI mode only)

  A switch that displays the contents of the report in a text window in addition to writing the information to *fileName*.

Description

Use this command after performing analyze on the desired targets.

Analyzes the difference vectors for each target loaded into the debugger and writes information about the input/register value assignment percentages to the file *fileName*. TDVR (Target Difference Vector Report).

Examples

fpdebug> tdvr

fpdebug> tdvr -f diff_vector.txt

whatif

Modifies the functionality of the difference network based on the *whatifArgument*

Usage

whatif *whatifArgument*

Arguments

- **convert *gateType*** {***gateName*** …}

  An argument that converts all *gateName*(s) to the specified *gateType*.

  gateType — An argument that specifies a primitive gate type, which is one of the following:

  buf, **not**, **and**, **nand**, **or**, **nor**, **xor**, **xnor**, **mux**

  gateName — An argument that specifies the name of a gate. You can specify any number of names for this argument.

  The following example shows how the **whatif convert** command converts a standard Verilog AND statement to a NOR statement. Given the statement:

  ```
  and (\B.n51625 , \B.data0 , \B.sel );
  ```

  the debugger command:

  ```
  fpdebug> whatif convert nor \B.n51625
  ```

  changes the above statement to the following:

  ```
  nor (\B.n51625 , \B.data0 , \B.sel );
  ```

- **invert** {***gateName*** …}

  An argument that inverts all *gateName*(s).

  gateName — An argument that specifies the name of a gate. You can specify any number of names for this argument.

- **const0** {***gateName*** …}

  An argument that ties all *gateName*(s) to the constant value of 0.

  gateName — An argument that specifies the name of a gate. You can specify any number of names for this argument.

- **const1** {***gateName*** …}

  An argument that ties all *gateName*(s) to the constant value of 1.

  gateName — An argument that specifies the name of a gate. You can specify any number of names for this argument.

- **deleteport *gateName*** {***portName*** …}

  An argument that deletes the connection of all nets in the *portName* list from the gate *gateName*.

gateName — An argument that specifies the name of a gate.

portName — An argument that specifies the name of a port (net name). You can specify any number of names for this argument.

The following example shows how the **whatif deleteport** command transforms a standard Verilog statement. Given the statement:

```
or (\B.select , \B.data0 , \B.data1 , \B.data2 , \B.data3 );
```

the debugger command:

```
fpdebug> whatif deleteport \B.select \B.data3
```

changes the above statement to the following:

```
or (\B.select , \B.data0, \B.data1 , \B.data2 );
```

- **invertport** ***gateName*** {***portName*** …}

  An argument that inverts the connection of all nets in the *portName* list into the gate *gateName*.

  gateName — An argument that specifies the name of a gate.

  portName — An argument that specifies the name of a port (net name). You can specify any number of names for this argument.

  The following example shows how the **whatif invertport** command transforms a standard Verilog AND statement:

  Given the statement:

```
and (\B.n51625 , \B.data0 , \B.sel );
```

  the debugger command:

```
fpdebug> whatif invertport \B.n51625 \B.sel
```

  changes the above statement to the following:

```
not (\B.sel_fpdebugbar , \B.sel );
and (\B.n51625 , \B.data0 , \B.sel_fpdebugbar );
```

- **addports** ***gateName*** {***portName*** …}

  An argument that adds the connection of all nets in the *portName* list into the gate *gateName*.

  gateName — An argument that specifies the name of a gate.

  portName — An argument that specifies the name of a port (net name). You can specify any number of names for this argument.

  The following example shows how the **whatif addports** command transforms a standard Verilog statement. Given the statement:

```
or ( \B.select , \B.data1, \B.data2 , \B.data3 );
```

the debugger command:

```
fpdebug> whatif addports \B.select \B.data0
```

changes the above statement to the following:

```
or ( \B.select , \B.data1 , \B.data2 , \B.data3 , \B.data0);
```

- **replaceports** ***gateName*** {***portName*** …}

  An argument that replaces the net connection into the gate *gateName*, with those in the *portName* list.

  gateName — An argument that specifies the name of a gate.

  portName — An argument that specifies the name of a port (net name). You can specify any number of names for this argument.

  The following example shows how the **whatif replaceports** command transforms a standard Verilog statement. Given the statement:

```
or ( \B.mmp0.enable , \B.mmp1.proc1_en , \B.mmp1.proc2_en );
```

  the debugger command:

```
fpdebug> whatif replaceports \B.mmp0.enable \B.mmp0.proc1_en \
B.mmp0.proc2_en
```

  changes the above statement to the following:

```
or ( \B.mmp0.enable , \B.mmp0.proc1_en , \B.mmp0.proc2_en );
```

- **build** [*fileName*]

  An argument that selects the **whatif** build mode, which allows you to specify multiple **whatif** modifications. The debugger stores all **whatif** commands specified after the **whatif build** command in an internal buffer.

  fileName — An argument that specifies a file to be loaded into the buffer for the build mode.

  After entering the whatif build command, the prompt changes to whatif>.

- **list**

  An argument that lists all commands in the current buffer.

- **save** ***fileName***

  An argument that writes the current buffer to the file *fileName*.

  fileName — An argument that specifies the name of the file to be written.

- **read** ***fileName***

  An argument that reads in the file *fileName* and concatenates it into the current buffer.

  fileName — An argument that specifies the name of the file to be read.

- **check** [-stopOnDiff *number*] [*target* …]

  An argument that applies the changes specified in the buffer to the network and checks the network for equivalence.

  -stopOnDiff *number* — A switch that stops the check after *number* differences are discovered.

  target — An argument that checks the effect of the modifications in the buffer on all targets in the *target* list. You can specify this argument any number of times in the command line.

- **end**

  An argument that ends the build mode and clears the buffer.

Description

Internally modifies the functionality of the difference network, based on the *whatifArgument*, then queries the difference network to determine the equivalency of the affected target(s).

You can perform *whatif* analysis in two ways: either as a single modification with immediate results, or as a series of modifications (build mode) that allows for incremental checking.

Note

The whatif command does not provide a permanent change. Essentially, it modifies, checks status, and restores to the original network.

Many of the whatif commands require either *gateName* or *portName* arguments, or both. The *gateName* and *portName* arguments are strings that specify a gate instance name in the difference network (such as U123) or a full path net name (for example \A.top.U1.mysub.net2). You can reference a *gateName* by its instance name or the net it drives. You can specify a *portName* as an input port of a gate that is referenced by the net attached to this port or the gate driving this net.

Caution

Typically, the design loaded into the debug environment is a subset of the entire design. When you alter a design, based on results from FormalPro, it is possible that modifications that create equivalency in this subset may cause inequivalencies in the portion of the design existing outside of the subset. You should always rerun FormalPro to verify all modifications for the entire modified design.

Like any other debug commands, you can type **whatif** commands at the **fpdebug** prompt. In GUI mode, you can access all **whatif** commands, except addports and replaceports, from a popup menu in the schematic.

To use **whatif** from the GUI for commands that require several arguments, first select the nets to be passed as an argument. Then right-click on the gate to modify and choose the relevant

whatif item in the popup menu. For example, to delete a port on a cell, first select the net driving the port, then right-click on the cell to select the **whatif > delete port** command.

Chapter 5
Input File Syntax

Depending on your need, there are several types of input files you can alter/create to specify how parts of the design files are interpreted during checking.

- Constraint and Match File Scripts
- Rule Files — Supply regular expression replacement and substitution rules to implicitly match comparison points between the two designs.
- Match Files — Specify explicit matches between comparison points in the two designs.
- Black Box Files — Declare Verilog modules or VHDL entities as black boxes.
- Constraint Files — Control how FormalPro compiles, matches and solves the two designs.
- Configuration Files — Apply design constraints at the compile stage.

You can get help information on these files from the command line by typing the following command at the shell prompt:

```
formalpro -help [rules | match | constraints | blackbox]
```

Constraint and Match File Scripts

Within Match and Constraint files, you can specify arbitrary scripts in any language, such as Perl or Tcl, that specify a large number of matches or constraints.

For example, suppose a 32-bit bus was reversed during synthesis, such that **buss(0)** and **buss(1)** in design A correspond to **buss[31]** and **buss[30]** in design B, respectively. Instead of writing 32 explicit match commands you could use the source command, as shown in the following example:

```
source $FORMALPRO_HOME/bin/fp_tclsh
for {set i 0} {$i < 32} { incr i } {
   puts "match register \\A.top.buss\($\{i\}\) \\B.top.buss\[[expr 31-$i]\]"
}
endsource

#Note that the leading '\' must be escaped.
#Note that it is best to use {} around your variable names to avoid
confusion
```

where source and endsource are the commands for this functionality, and */usr/local/bin/fp_tclsh* is the executable used to run the script located between source and endsource. If you use the $FORMALPRO_HOME environment variable, you should also specify it in your source command.

The usage syntax for this functionality can take two forms:

```
source <executable>
<body>
endsource
```

```
source
#!<executable>
<body>
endsource
```

where FormalPro uses the language specified in <executable> to run the script within <body>. FormalPro replaces, inclusively, all lines between source and endsource, with the standard output of <executable>. Once this is completed, FormalPro resumes processing the Constraint or Match file as normal.

VHDL Read Order File

Within the **library** category, you can specify an *order file* that appends the default order file in the library directory. An order file is a text file that specifies the order in which FormalPro reads VHDL library files. Each VHDL library shipped with FormalPro contains a default order file (.ord).

You can specify your own order file by listing its location after the library location, separated by a colon. For example:

```
[library]
ieee        = $FORMALPRO_HOME/lib/fp_vhdl_libs/ieee:$HOME/myieee.ord
```

Options Applied Based on Platform

You can control which initialization file options are applied based on the platform type with #ifdef, #ifndef, #else, #endif conditional operators. The conditional blocks cannot be nested and platform options include "windows" and "linux".

For example:

```
stopOnUnmatched   = false
#ifdef WINDOWS
mp                = 1
#else
mp                = 4
#endif
mplimit           = 16
```

Rule Files

A rule file is a list of user-defined replacement rules that FormalPro uses during the match stage to facilitate name matching of ports and registers in one design with the ports and registers of the other. The replacement rules consist of regular expressions plus additional syntax described in the “Usage” section below.

The affect of applying the replacement rules is accumulative. The names are transformed according to the first rule and then an attempt to match A-side names with B-side names. Any remaining unmatched names are then further transformed according to the second rule and another attempt to match names is made. This is repeated until all ports and registers are matched or all match rules have been applied.

The order of the replacement rules affects performance during the match phase. You should order the rule list with the rules likely to match the most names first. This reduces the pool of unmatched names faster, shorten the processing time of each subsequent match attempt.

Rule file syntax supports “rule sets”. At the end of a rule set, FormalPro resets any unmatched names to their original state before applying the next set of rules. To define the end of a rule set, insert a line in the rule file containing only a single period (.).

Comment lines in the rule file start with the pound sign (#) and end with a newline. FormalPro also treats as a comment any line starting with a space, tab, or newline.

User-Defined Rule Files

By default, FormalPro uses the following rule file containing some basic replacement rules:

$FORMALPRO_HOME/pkgs/fv/userware/rules.default

There are various ways to replace or augment the default rules. You can:

- Modify the default rule file
- Specify a user-defined rule file (the -ruleFile switch)
- Specify additional user-defined rule files (the -addRuleFile switch)

The **-ruleFile** switch enables you to specify a user-created rule file that is used in place of the default file. The default rule file is ignored. Also ignored is any rule file specified in a previous occurrence of the -ruleFile switch or the -addRuleFile switch (for example, if you specify one of the switches in a formalpro.ini file). When creating your own rule file, use these guidelines:

- Because the default rules are ignored when you use the -ruleFile switch, it is typically best to start with a copy of the default rules and modify or augment them as needed.
- Do not save user-defined rule files in the FormalPro cache because the cache is overwritten in subsequent runs.

The **-addRuleFile** switch enables you to specify multiple rule files. The rules in multiple files are applied in the order that the files are specified. For example, assume the switches appear in this order:

```
-ruleFile A  -addRuleFile B  -addRuleFile C
```

In this case, the default rule file is ignored, the rules in A are applied, then the rules in B, then the rules in C.

The occurrence of the -ruleFile switch completely overrides the operative replacement rules. For example, assume the switches appear in this order:

```
-ruleFile A  -addRuleFile B  -ruleFile X -addRuleFile C
```

In this case, the rules in X are applied, then the rules in C. The rules in A and B are overridden by the occurrence of -ruleFile X.

Consider the case where -addRuleFile is used, with no preceding -ruleFile:

```
-addRuleFile B
```

In this case the rules in the default rule file are applied, then the rules in B.

Replacement Rule Usage

designId [+ | -] [p | i | o | r | f | l] ***rule***

Arguments

- ***designId***

  A required string, which must be “A” or “B” (uppercase is required), indicating the design to which the rule applies.

  When specifying any of the following arguments, you cannot separate the arguments from the designId by any white space. The following examples show correct usage of the arguments:

  ```
  A-  s/alpha/beta/
  Ap  s/alpha/beta/
  Ap+ s/alpha/beta/
  Ai- s/alpha/beta/
  ```

- +

  An argument that allows names that are the same in one design, due to name transformation based on rules, to be matched to a single name in the other design. Normally, only a single name from design A can be matched to a single name in design B.

- -

  An argument that states that no match analysis is to be performed after transforming the names based on the rule. This allows you to perform multiple changes on a name before attempting matching.

- p | i | o | r | f | l

  Argument that applies the rule only to a specific type of object.

p	ports	r	registers
i	inputs	f	flip-flops
o	outputs	l	latches

- ***rule***

  A replacement expression conforming to the syntax of the **s**, **y**, and **d** commands of the sed editor as shown:

  Substitution — performs string substitution on the first occurrence of original.

  s/original/replacement/flag

where flag can be:

g — replace all occurrences of original

n — replace the nth occurrence of original. n must be between 1 and 152.

Translation — performs character translation

y/string1/string2/

Deletion — disregards objects for matching, whose names contain a given string

/string/d

Examples

The following transformation rule transforms all uppercase characters in design A to lowercase:

```
A y/ABCDEFGHIJKLMNOPQRSTUVWXYZ/abcdefghijklmnopqrstuvwxyz/
```

The following substitution rule replaces, in design A, the string "_*i*_" at the end of a line by the string "_b*i*", where *i* is any number.

```
A s/_\([0-9]*\)_$/_b\1/
```

The following substitution rule suppresses the first occurrence of the string "_reg" in all targets of design B.

```
B s/_reg//
```

The following substitution rule replaces, in design B, all occurrences of the string "clk_*x*_*y*" with the string "clk", where *x* and *y* are any number, and allows multiple matching (as specified by the plus sign (+)). This rule would be useful for matching an external clock tree.

```
B+ s/clk_[0-9]*_[0-9]*/clk/g
```

The following substitution rule disregards names for matching, such as internally generated names that end in ".n*i*" in design A.

```
A /\.n[0-9]*/d
```

Match Files

A match file allows you to specify explicit matches of comparison points between designs A and B. Use the **-matchFile** command line switch to load a match file into FormalPro.

In the match file, comments start with the pound sign (#) character and end with a new line.

Within the Match file, you can use the Source command to include scripts that create match commands. Refer to the section "Constraint and Match File Scripts" on page 206 for further information.

Complement Matching

FormalPro allows you to explicitly match two objects that have different phases. You can make this type of match using the *match_compl* command in place of match.

The action of the *match_compl* keyword is similar to that of the *match* keyword, except that *match_compl* establishes a direct correspondence between an object in design A (*ObjRefA*) and the invert of an object in design B (*ObjRefB*).

For each of the command syntax examples in the following sections, you can replace the match keyword with match_compl.

Matching Registers

This section describes the match command for matching registers, which include both DFF and latch devices.

Usage

- Register

  match register [options] fullRegNameA fullRegNameB

You can substitute match with match_compl to perform complement matching between the registers.

Arguments

- fullRegName — A full-path register name.
- options — The options are described in the section "Match Command Options".

Examples

```
match register \A.top.foo_cpu.reg1 \B.top.bar_cpu.reg1
match register \A.top.foo_bus[31:0] \B.top.foo_bus[0:31]
match register \A.top.array[0:64][7:0] \B.top.offsetArray[8:72][8:1]
```

Matching Objects of Top-level Modules

This section describes the match commands for matching objects of top-level modules between designs A and B.

Usage

- Input ports

  match input [options] fullPortNameA fullPortNameB

match input [options] A portNameA B portNameB

- Output ports

  match output [options] fullPortNameA fullPortNameB

  match output [options] A portNameA B portNameB

- Bidirectional ports

  match bidir [options] A portNameA B portNameB

- Top-level module

  match top [options]

You can substitute match with match_compl to perform complement matching between the top-level module objects.

Arguments

- fullPortName — The name of a port object from a top-level module. This object must appear in the scalar port file as an input.
- portName — The name of a top-level port on a design. This variable can appear as follows:

  in \in \in[0] in[0] in [0] \in [0]

- options — The options are described in the section "Match Command Options", where -force may be the most useful.

Examples

```
match input A in2[0] B in1[0]
match_compl input \A.top.in2[0] \B.top.in1[0]
match output A outA[0] B outB[0]
match output \A.top.in2[0] \B.top.in1[0]
```

The following example matches the ports of the top-level modules of designs A and B by their port position, rather than their name:

```
match top -by_position
```

Matching Objects of Black Box Instances

This section describes the match commands for matching objects of an instance that was declared a black box.

Usage

- Input ports

match bbinput [options]
fullPortNameA fullPortNameB

match bbinput [options]
instNameA portNameA instNameB portNameB

- Output ports

atch bboutput [options] fullPortNameA fullPortNameB

match bboutput [options] instNameA
portNameA instNameB portNameB

- Bidirectional ports

match bbbidir [options]
fullPortNameA fullPortNameB

match bbbidir [options]
instNameA portNameA instNameB portNameB

- Black Box instance

match bbinst [options]
instNameA instNameB

You can substitute match with match_compl to perform complement matching between the black box instance ports.

Arguments

- fullPortName — The name of a port object from a top-level module. This object must appear in the scalar port file as an input.
- instName — A full-path instance name.
- portName — The name of a top-level port on a design. This variable can appear as follows:

in \in \in[0] in[0] in [0] \in [0]

- options — The options are described in the section "Match Command Options", where -force may be the most useful.

Examples

```
match bbinst \A.top.bb_instance.in1 \B.top.bb_inst.in1
```

Match Command Options

The option argument allows you further control of the explicit match command. The selected defaults are conservative in that they allow a match to be made only when it is very likely to be correct. This conservative choice is for matching vectored ports, in particular. When you want to match two scalar ports, you should use the -force switch.

- -force — An argument that is useful when matching two scalar ports.

The default behavior (not specifying **-force**) is conservative, which means a match is made only when it is very likely to be the right one. You should use the default behavior when matching vectored ports, in particular.

An alias for this command is **-f**.

This option applies only to the *types:* **input**, **output**, **bidir**, **bbinput**, **bboutput**, or **bbbidir**.

- -by_name | -by_position — Matches port, at the module or instance level, by name or by position.

-by_name — Designates two ports as potentially matchable if they have the exact same name. Both ports must appear in the scalar port file as an input. This syntax is convenient when manually matching by copying names directly from the unmatched report.

-by_position — Designates the nth port of the first module instance as potentially matchable to the nth port of the second module instance. For example, if there are five ports in the first module instance and three ports in the second module instance, there are three potentially matchable ports.

This option applies only to the *types:* **top or bbinst**.

- -all | -partial — Specifies how many of the ports of both module instances must be designated as potentially matchable for a match to occur.

-all — All ports must be designated as potentially matchable. Otherwise, none of the ports will be matched, and the set of potentially matchable pairs is set to empty.

-partial — Only a subset of the ports must be designated as potentially matchable.

This option applies only to the *types:* **top or bbinst**.

- -by_bit_position | -by_bit_index — Specifies that bits are designated as potentially matchable, dependent on their bit position or their bit index value.

-by_bit_position — Designates two bits as potentially matchable if they have the same lsb to msb position.

-by_bit_index — Designates two bits as potentially matchable if they have the same index value.

- -exact_width | -diff_width — Specifies that bits are designated potentially matchable, dependent on the width of the two ports.

-exact_width — Designates that two bits are potentially matchable if the widths of the two ports are equal. Otherwise, the set of potentially matchable bit pairs is set to empty.

-diff_width — Designates that the set of potentially matchable bit pairs is not affected by the width of the ports.

- -no_priors | -priors_ok — Specifies that bits are designated potentially matchable, dependent on whether they have been matched previously.

-no_priors — Designates that none of the bits can be matched previously. Otherwise, the set of potentially matchable bit pairs is set to empty.

-priors_ok — Designates that the set of potentially matchable bit pairs is not affected by previous matching.

- -no_multiple | -multiple_ok — Specifies that bits are designated potentially matchable, dependent on whether either bit of the pair has been matched previously.

-no_multiple — Designates that two bits are matched only if neither of them are already matched.

-multiple_ok — Designates that two bits are matched regardless of whether one or both of them are already matched.

Black Box Files

A black box file enables you to black box and partition the design. The encapsulate commands insert compare points at module boundaries to help partition the design. The dpAddGroup command enables the grouping of multiple modules and the addition of compare points at the boundaries of the group.

The **-blackboxFile** command line switch specifies which black box files get loaded.

blackbox, encapsulate, and noencapsulate

FormalPro reads, but does not compile, a black-boxed object, with the exception of its port definitions and declarations (Verilog) or its component declaration (VHDL). FormalPro automatically infers and treats a black-boxed object if it does not exist but is instantiated in the design. FormalPro ignores an object listed in the black box file if it does not exist in the design.

Encapsulation creates comparison targets on the ports of a module and isolates that module from the rest of the design. Both the encapsulated module and the rest of the design are solved but the connection between them is severed. This enables you to verify modules in place, thus avoiding separate verification steps.

Note

FormalPro can only encapsulate user-defined modules. Modules that are read in from libraries and modules that are generated by the RTL compiler cannot be encapsulated.

The encapsulate and blackbox commands can be applied to all modules or only to a specific elaboration of a module. You can use a wildcard character (*) to specify a group of modules. For example, to encapsulate all user-defined modules, use the following commands:

```
encapsulate A *
encapsulate B *
```

You can also use the **noencapsulate** command in the blackbox file. This command prevents encapsulation of the specified modules. The **noencapsulate** command will always override encapsulate command or the **-encapsulateAll** switch. You can place the noencapsulate command either before or after the same module in the blackbox file and it will behave the same.

Comments in a blackbox file start with the pound sign (#) character and end with a new line. All other lines of this file must conform to the syntax shown under “Usage” below.

The encapsulate, noencapsulate, and blackbox commands all use the same syntax and take the same arguments. The one exception: only the blackbox command takes the instance path argument.

Syntax

blackbox ***{A | B} moduleName*** (Verilog format designs)

blackbox ***{A | B} libName.entName*(*archName*)** (VHDL format designs)

blackbox ***{A | B} moduleName***-parameters

blackbox ***{A | B} entName*(*archName*)**

blackbox ***{A | B} entName***

blackbox ***{A | B} libName-entName-archName***

blackbox ***{A | B} libName-entName***-generics-***archName***

blackbox ***{A | B} instance_path***

blackbox ***instance_path***

encapsulate ***{A | B} moduleName***

encapsulate ***{A | B} libName.entName*(*archName*)**

encapsulate ***{A | B} moduleName***-parameters

encapsulate ***{A | B} entName*(*archName*)**

encapsulate ***{A | B} entName***

encapsulate ***{A | B} libName-entName-archName***

encapsulate ***{A | B} libName-entName***-generics-***archName***

Arguments

You can use wildcards (*) in all of the following arguments except for *instance_path*

- ***moduleName***

  An argument that specifies the name of the module to black box. This argument only applies to Verilog format designs.

- ***libName***

An argument that specifies the name of the VHDL library containing the entity to black box. This argument only applies to VHDL format designs. This argument must be followed by a period (.).

- ***entName***

  An argument that specifies the name of the entity to black box. This argument applies only to VHDL designs. This argument must follow the *libName* argument and a period (.).

- (***archName***)

  An argument that specifies the name of the architecture to black box. This argument applies only to VHDL designs. You must enclose this argument in parentheses and specify it following the *entName* argument.

- ***instancePath***

  An argument that specifies the name of an instance path to a module. The format of the instance path is the standard FormalPro format (for example, \A.top.inst1.inst2). Wildcards are not permitted.

  Note that the second character in an instance path indicates the design side and the A/B side argument can be omitted. However, if you include it, it must match the design side indicated in the path.

  The design side argument is still allowed but it is optional. If it exists, it must match the side indicated in the instance path.

Examples

Black box the modules *ram128x16* and *\RAM_#23* in design A (Verilog):

```
blackbox A ram128x16
blackbox B \RAM_#23
```

Black box the entity(architecture) foo(foo_rtl) in library mylib of design B (VHDL):

```
blackbox B mylib.foo(foo_rtl)
```

Using wildcards, black box multiple RAM modules with different names (RAM_128, RAM_256, and RAM_address_decoder):

```
blackbox B RAM_*
```

Using the instance format for black box:

```
blackbox A \A.top.inst1.inst2
blackbox \B.top.insta.instb
```

dpAddGroup

Include this datapath manual grouping command in a black box file to group together a collection of design instances into a datapath, overriding the automatic grouping feature. When any manual grouping is being used, no automatic groups are created.

Syntax

dpAddGroup ***{Tcl list of specification instance names} {matching implementation instance name}***

Arguments

- ***{Tcl list of specification instance names}***

  Specifies a Tcl list of specification instance names that must be fully qualified names. This list is paired with the specified implementation instance.

  The number of groups is unlimited.

  When the -dataPath option is used without the dpAddGroup commands, the groups are automatically made and are listed in the multarch.report. The syntax below can be copied from the report to create a manual subset of the automatic results (recommended).

- ***{matching implementation instance name}***

  Specifies the matching implementation instance name that is paired with specification instance names.

Examples

The TCL list for design *A* contains instance names separated by spaces. That collection maps to one datapath module in the design *B* which might be an example of a multiply-add.

```
dpAddGroup { \A.Dwidth_JV32.rtlc0M_297  \A.Dwidth_JV32.item2  }
   { \B.Dwidth_JV32.i_5_12 }
```

Constraint Files

A constraint file allows you to apply constraints to objects within each design in order to reduce processing time or to ignore or bypass aspects of the designs. You can load one or more constraint files into FormalPro with the **-constraintFile** command line switch or by entering the filenames in the Constraint field in the GUI.

The constraints are applied during the match stage of the equivalence checking session.

The Source command can be used in a constraint file to include arbitrary scripts that create constraint commands. For more information, see “Constraint and Match File Scripts” on page 206.

You can automatically create a constraint file from within the FormalPro GUI, either from the Match Tool Window or from the Target List Window.

Tip

If you have a set of constraints used on multiple designs, you can place them in a master constraint file and reference it by adding the following statement to your *formalpro.ini* file:
constraintFile = master.constr

The following topics describe commands that can be used in a constraint file:

assert

Scope: Constraint file

Verifies a specified register or net in a design evaluates to a constant.

Upon receiving an assertion constraint, Formalpro creates an assertion comparison point (or target) that checks that the function specified at the register or net evaluates to the constant. In addition, it assumes that the assertion holds true for any logic fed downstream of the function, such that any other comparison point or target fed by the asserted register or net will be driven by the specified assertion constant value. This constraint is particularly useful for ensuring that the constant optimizations performed by the synthesis tools are correct.

When assertion points are added, the formalpro log reports the number of assertion point failures that were uncovered. For example, the following excerpt from a log file shows that 39 assertion failures were uncovered.

```
Comparison Summary
==================
Total number of comparison points: 34783
--------------------------------------------------
Number of Equivalent comparison points: 34744
--------------------------------------------------
Fed by unmatched net 44
Number of Different comparison points: 39
--------------------------------------------------
Assertion failures 39
```

All assertion comparison points are created with a target names that have a prefix consisting of the string “ta” followed by a numeric value (for example, ta321123). Like design comparison points, assertion comparison points can be viewed within the target table display and passed into FormalPro's debug environment. However, they cannot be viewed in the design hierarchy schematic browser.

Note

The behavior of the assertion constraint is somewhat similar to the force constraint except that it creates a comparison point to check the assertion, whereas the force constraint does not.

Syntax

assert register {0 | 1} <full_instance_path_name_of_register>

assert net {0 | 1} <full_instance_path_name_net>

Examples

```
assert register 0 \A.top.modInst1.shutdown_reg
assert net 0 \A.top.ucore.bist_controller.run_bist
```

complement

Scope: Constraint file

Specifies a complement polarity to registers or ports in your designs and libraries.

By default, a positive polarity is assigned to all canonical names of the registers, with the polarity of the register aliases based on that polarity. This constraint changes the default behavior and assigns a negative polarity to the canonical name of the register in the database.

This constraint is most useful for library cells that use registers with Q-bar outputs, which allows for easier matching.

Syntax

- Individual register

  complement register fullRegName

- Every register in a module

  complement module {A | B} moduleName

- Top-level ports

  complement input fullPortName

  complement input {A | B} portName

  complement output fullPortName

  complement output {A | B} portName

  complement bbinput fullPortName

  complement bbinput instName portName

  complement bboutput fullPortName

  complement bboutput instName portName

Arguments

- fullRegName — A full-path register name.
- moduleName — The name of a module or entity.
- instName — A full-path instance name.
- fullPortName — The name of a port object from a top-level module. This object must appear in the scalar port file as an input.
- portName — The name of a top-level port on a design. This variable can appear as follows:

  in \in \in[0] in[0] in [0] \in [0]

Examples

```
complement register \A.top.foo_reg
complement module A libcell_dff
complement input \A.top.clrn
```

duplicate and duplicate_compl

Scope: Constraint file

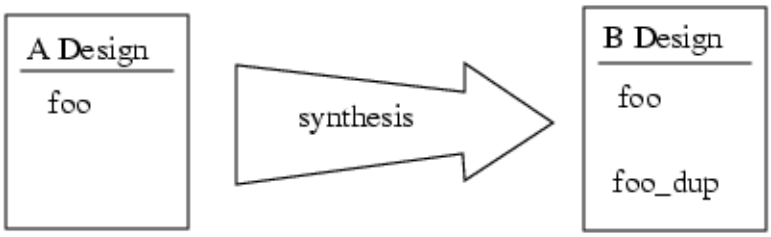
Identifies/resolves duplicate ports and registers.

The synthesis process can produce asymmetry in registers and ports between the A design and B design. For example, for a single register in the A design, synthesis may generate two registers in the B design, or for two registers in the A design, synthesis may generate a single register in the B design.

During the match phase when FormalPro forms targets by matching registers/ports, this asymmetry may result in unmatched objects. The **duplicate** command explicitly identifies duplicated objects, so they can be resolved.

Consider the case in which two registers in B (foo and foo_dup) are created from a single register in A (foo) shown in Figure 5-1.

Figure 5-1. Register Asymmetry Introduced by Synthesis

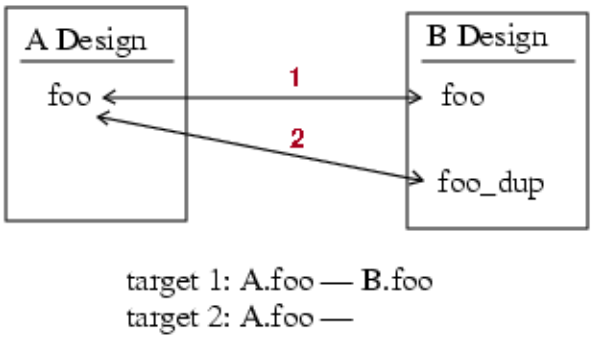


The following **duplicate** command in the constraint file facilitates matching by specifying that in design B, foo_dup is a duplicate of foo:

```
duplicate B foo foo_dup
```

In the match phase FormalPro produces two targets that are checked for equivalence in the solve phase.

Figure 5-2. Targets Produced by Duplicate Matching



Note

Redundant detection and register/port merging can also work in reverse. For example, if A has two identical registers, then they are redundant and can be merged for area reduction with a 'duplicate A' statement.

duplicate_compl is similar to **duplicate**, except it applies to duplicates that are the complement of the register it duplicates. For example:

```
duplicate_compl B foo foo_dup
```

duplicate_compl identifies foo_dup in the B design as an inverted duplicate of foo.

The **duplicate** command syntax supports wildcards and various means of identifying duplicates that are instantiated multiple times (see the examples below).

Syntax

duplicate [port] ***fullregister_portName1 fullregister_portName2***

duplicate [port] {A | B} ***moduleName.register_portName1 moduleName.register_portName2***

duplicate_compl [port] ***fullregister_portName1 fullregister_portName2***

duplicate_compl [port] {A | B} ***moduleName.register_portName1 moduleName.register_portName2***

Arguments

- *port*

  Specifies that both object names are ports. Duplicate ports can be primary inputs/outputs or blackbox input/outputs. A **duplicate** command sets up a check to prove that the duplicate is true along with facilitating the matching process where the removal of one of the items in the revised netlist will not upset the match pairings. By default, object names are assumed to be registers.

- *fullregister_portName1*, *fullregister_portName2*

  Specifies the full register/port name. This format applies only to the registers/ports specified by *fullregister_portName1* and *fullregister_portName2*. Example (Verilog):

  ```
  duplicate \B.ucbf_top.us.reg_buf0_20_ \
  B.ucbf_top.u4.buf0[20]_DUPLICATE_I
  ```

- *moduleName.register_portName*

  Specifies the name of the register/port and the parent module. This format applies to every duplication of the register in all instantiations of the module. In this format, the register name does not include the design indicators, so you must use the {A | B} option in the command line. In this format, the previous example would look like:

  ```
  duplicate B usbf_top.middle.buf0(20)  usbf_top.middle.buf0_copy(20)
  ```

Examples

Example 1

In the following VHDL example, the register reg_current(2) is identified as a duplicate of reg_current_state(1) in every instantiation of work-dma_engine_ent-dma_engine_arc in design B.

```
duplicate B work-dma_engine_ent-dma_engine_arc.reg_current_state(2)
work-dma_engine_ent-dma_engine_arc.reg_current_state(1)
```

Example 2

The following example uses the fullregisterName format; it applies only to a single instantiation of a duplicated register, in module fred, in design A. Register foo_dup is identified as a duplicate of foo.

```
duplicate \A.top.fred.foo \A.top.fred.foo_dup
```

Example 3

The following example uses wildcard matching. All registers in module fred that match foo(*) are identified as duplicates of fred(0).

```
duplicate \A.top.fred.foo(0) \A.top.fred.foo(*)
```

Example 4

In the following example, for every instantiation of module fred in design A, all registers that match foo(*) are identified as duplicates of fred(0).

```
duplicate A fred.foo(0) fred.foo(*)
```

Example 5

The following examples demonstrate how the complement characteristic propagates when the **duplicate**, **complement**, and **duplicate_compl** commands are used together. NOTE: The order of commands affect the propagation of complement characteristics.

Assume you need to identify a set four registers: 2 duplicates and 2 duplicate complements (reg1, reg2, ~reg3, ~reg4).

The following sequence of commands show the least error-prone way to accomplish the desired set: first use the **duplicate** command to create the set of duplicates, then identify the complement registers in the set.

```
duplicate \A.top.reg1 \A.top.reg2 #(reg1, reg2)
duplicate \A.top.reg1 \A.top.reg3 #(reg1, reg2, reg3)
duplicate \A.top.reg1 \A.top.reg4 #(reg1, reg2, reg3, reg4)
complement register \A.top.reg3   #(reg1, reg2, ~reg3, reg4)
complement register \A.top.reg4   #(reg1, reg2, ~reg3, ~reg4)
```

The following sequence of commands illustrate the unexpected results when reg3 and reg 4 are first identified as complements, then subsequently identified as duplicates of a register. The complement characteristics assigned in the first two commands are overridden by the last two **duplicate** commands. In the end, none of the registers in the set are complements.

```
complement register \A.top.reg3    # ~reg3
complement register \A.top.reg4    # ~reg4
duplicate \A.top.reg1 \A.top.reg2  # (reg1, reg2)
duplicate \A.top.reg1 \A.top.reg3  # (reg1, reg2, reg3)
duplicate A.top.reg1 \A.top.reg4   # (reg1, reg2, reg3, reg4)
```

Command 1 tells FormalPro, “in the match phase, match the complement of reg3 (A design) with reg 3 (B design). But command 3 contradicts this matching, implicitly matching reg3 (A design) with reg1 (B design). In effect command 3 says, “in A design reg3 is a duplicate of reg1, therefore match reg3 in the A design with *reg1* in the B design”.

The following sequence of commands yields the desired results using the duplicate_compl command:

```
duplicate_compl \A.top.reg1, \A.top.reg3   #(reg1, ~reg3)
duplicate \A.top.reg1 \A.top.reg2          #(reg1, reg2, ~reg3)
duplicate \A.top.reg3 \A.top.reg4          #(reg1, reg2,~reg3, ~reg4)
```

The following sequence of commands also yields the desired results using the duplicate_compl command:

```
duplicate_compl \A.top.reg1 \A.top.reg3  #(reg1, ~reg3)
duplicate_compl \A.top.reg2 \A.top.reg4  #(reg2, ~reg4)
duplicate \A.top.reg1 \A.top.reg2        #(reg1, reg2, ~reg3, ~reg4)
```

Example 6

The following command sets up duplicate ports for verification:

```
duplicate port \A.top.Out_port \A.top.In_port
#(Prove that Out_port is equivalent to In_port, which is a pass-thru)
```

Related Commands

For a similar function that relates to ports, see the “tie and tie_compl” on page 234.

eco_correspond

Scope: Constraint file

Specifies that a pair of signals are equivalent points.

A functional correspondence is a pair of signals that define equivalent points in the ECO and original layout design failing regions (cones), such that the two designs are functionally equivalent if all these cuts are taken as primary inputs. Essentially, these cuts bound the patch region on the “output side” of the ECO logic.The FormalPro tool provides robust techniques to automatically find appropriate functional correspondence cuts. However, in certain cases user intervention can provide additional correspondence hints as constraints to the tool. These hints help in the quality of the eventually generated ECO patch.

Syntax

- **eco_correspond** ***fullNetName fullNetName***

Examples

```
eco_correspond \A.control_operation.n_7 \B.control_operation.n_12
```

force

Scope: Constraint file

Forces objects to a specified value of either 1 or 0.

By forcing an object you alter how the internal databases are compiled (constant propagation and register pruning). During the solve stage, FormalPro assigns the value to the specific object. The forced objects become "benign" in FormalPro logs and report status.

Forcing Objects of Top-level Modules

Syntax

- Input ports

  force input {0 | 1} fullPortName

**force input {0 | 1} {A | B}
portName**

- Bidirectional ports

**force bidir {0 | 1} {A |
B} portName**

Arguments

- fullPortName — The name of a port object from a top-level module. This object must appear in the scalar port file as an input.
- portName — The name of a top-level port on a design. This variable can appear as follows:

in	\in	\in[0]	in[0]	in [0]	\in [0]

Examples

```
force input 0 A test_en
force input 0 \B.top.scan_in
```

Forcing Objects of Black Box Instances

Syntax

- Output ports

force bboutput {0 | 1} fullPortName

**force bboutput {0 | 1} instName
portName**

- Bidirectional ports

**force bbbidir {0 | 1}
fullPortName**

**force bbbidir {0 | 1}
instName portName**

- Every output of a black box instances

**force bbinst {0 | 1}
instName**

Arguments

- instName — A full-path instance name.
- fullPortName — The name of a port object from a black box instance. This object must appear in the scalar port file as an input.

- portName — The name of a black box instance port. This variable can appear as follows:

  in \in \in[0] in[0] in [0] \in [0]

Examples

```
force input 0 \B.top.scan_in
```

Forcing Objects of Black Box Modules

Syntax

- Output ports

  force o/bbm {0 | 1} {A | B} bbModuleName portName

- Bidirectional ports

  force b/bbm {0 | 1} {A | B} bbModuleName portName

- Every output of a black box module

  force bbmodule {0 | 1} {A | B} bbModuleName

Arguments

- bbModuleName — The name of a black box module or entity.
- portName — The name of a black box module port. This variable can appear as follows:

  in \in \in[0] in[0] in [0] \in [0]

Examples

```
force o/bbm 0 B ram16x32 input
force bbmodule 0 A ram
```

Forcing Registers

Syntax

- Individual register

  force register {0 | 1} fullRegName

- Every register in a specific instance

  force instance {0 | 1} instName

- Every instance of a defined register

force register_def {0 | 1} {A | B} moduleName definedName

- Every register in a specific module

force module {0 | 1} {A | B} moduleName

Arguments

- fullRegName — A full-path register name. You can specify wildcards (*) when using this argument.
- instName — A full-path instance name.
- moduleName — The name of a module or entity.
- definedName — The name of a defined register.

Examples

```
force register 1 \A.top.cpu.reg1
force instance 0 \A.top.cpu.I0
force register_def 1 B cpu foo_register
force module 0 B cpu
```

Forcing an Individual Net

Syntax

- Individual net

force net {0 | 1} fullNetName

Arguments

- fullNetName — A full-path net name.

Examples

```
force net 0 \A.top.cpu.ci
```

Forcing Multiple Net Instances

Syntax

- Force a net defined in a module or library cell, in all instances of the module or library cell in a given design

force module_net {0 | 1} {A | B} moduleName netName

Arguments

- moduleName — The name of a module or entity
- netName — A local name of a net defined in *moduleName*

Examples

```
force module_net 0 B DFF1 notify
force module_net 1 A mylib.foo(foo_rtl) vcc_net
```

ignore

Scope: Constraint file

Specifies objects to ignore.

These objects maintain their real/benign status in FormalPro reports and log files.

Ignoring Objects of Top-level Modules

Syntax

- Output ports

  ignore output fullPortName

  ignore output {A|B} portName

- Bidirectional ports

  ignore bidir {A|B} portName

Arguments

- fullPortName — The name of a port object from a top-level module. This object must appear in the scalar port file as an input.
- portName — The name of a top-level port on a design. This variable can appear as follows:

in	\in	\in[0]	in[0]	in [0]	\in [0]

Examples

```
ignore output B scanout
ignore output \B.top.scan_out
```

Ignoring Objects of Black Box Instances

Syntax

- Input ports

  ignore bbinput fullPortName

  ignore bbinput {A | B} instName portName

- Bidirectional ports

  ignore bbbidir fullPortName

  ignore bbbidir {A | B} instName portName

- Every input of a black box instances

  ignore bbinst instName

Arguments

- instName — A full-path instance name.
- fullPortName — The name of a port object from a black box instance. This object must appear in the scalar port file as an input.
- portName — The name of a top-level port on a design. This variable can appear as follows:

in	\in	\in[0]	in[0]	in [0]	\in [0]

Examples

```
ignore bbbidir \A.top.inst1 data
```

Ignoring Objects of Black Box Modules

Syntax

- Input ports

  ignore i/bbm {A | B} bbModuleName portName

- Bidirectional ports

  ignore b/bbm {A | B} bbModuleName portName

- Every input of a black box module

  ignore bbmodule {A | B} bbModuleName

Arguments

- bbModuleName — Name of a black box module or entity.
- portName — Name of a black box module port. This variable can appear as follows:

in	\in	\in[0]	in[0]	in [0]	\in [0]

Examples

```
ignore bbmodule A rom4k8_comp
```

Ignoring Registers

Syntax

- Individual register

  ignore register fullRegName

- Every register in a specific instance

ignore instance instName

- Every instance of a defined register

  ignore register_def {A | B} moduleName definedName

- Every register in a specific module

  ignore module {A | B} moduleName

Arguments

- fullRegName — A full-path register name. You can specify wildcards (*) when using this argument.
- instName — A full-path instance name.
- moduleName — The name of a module or entity.
- definedName — The name of a defined register.

Examples

```
ignore register \A.top.I0.reg1
ignore module A cpu
```

no_match

Scope: Constraint file

Prevents a specified register from being matched to another during verification.

The *no_match* command supersedes entries in a match file. For more information on match files, see “Match Files” on page 210.

Syntax

- Register

  no_match register fullRegName

Arguments

- fullRegName — A full-path register name. You can specify wildcards (*) when using this argument.

Examples

```
no_match register \A.top.foo_reg
```

transparent

Scope: Constraint file

Converts registers in a design into buffers.

Use this command to compare designs with pipelined circuitry or for designs where latches occur due to gate-clocking. FormalPro reports transparent registers as “benign” unmatched objects.

Making Registers Transparent

Syntax

- Individual register

  transparent register fullRegName

- Every register in a specific instance

  transparent instance instName

- Every instance of a defined register

  transparent register_def {A | B} moduleName definedName

- Every register in a specific module

  transparent module {A | B} moduleName

Arguments

- fullRegName — A full-path register name. You can specify wildcards (*) when using this argument.
- instName — A full-path instance name.
- moduleName — The name of a module or entity.
- definedName — The name of a defined register.

Examples

```
transparent register \A.top.cpu.reg1
transparent instance \A.top.cpu
transparent module A mult
transparent register_def A cpu_def ir
```

tie and tie_compl

Scope: Constraint file

Shorts two or more ports together, where tie applies to ports that have the same logical value, and tie_compl applies to ports that have opposing logical values.

This functionality is useful for designs that have non-overlapping clock systems, as well as LSSD designs using a Master/Slave configuration.

Syntax

Note

There are additional conventions that must be followed to prevent errors in the match stage. For details, see the examples in this section.

- Input ports

  tie input fullPortName1 fullPortName2

  tie_compl input fullPortName1 fullPortName2

- Bidirectional ports

  tie bidir fullPortName1 fullPortName2

  tie_compl bidir fullPortName1 fullPortName2

- Output ports of black box instances

  tie bboutput fullPortName1 fullPortName2

  tie_compl bboutput fullPortName1 fullPortName2

- Top-level primary output to top-level primary input

  tie fullPortNameInput fullPortNameOutput

  tie_compl fullPortNameInput fullPortNameOutput

  Use this when verifying a sub-module of a design where an output port of a module feeds back into the same module as an input port. The output port is considered as a matchable comparison point, but not the input port.

Arguments

- fullPortName — The name of a port object from a top-level module. This object must appear in the scalar port file as an input.
- portName — The name of a top-level port on a design. This variable can appear as follows:

  in \in \in[0] in[0] in [0] \in [0]

Examples

In this example the first statement ties together two inputs in the A-design, the second ties together the corresponding inputs in the B design. FormalPro will create a solve target by matching the tied ports in the A-design with the tied ports in the B-design.

```
tie input \A.top.dclk_one_ram \A.top.dclk_one
tie input \B.top.dclk_one_ram \B.top.dclk_one
```

Note that in both tie statements, **dclk_one_ram** is on the left. This convention must be followed to facilitate matching. If the order is reversed in the second statement, as shown below, Formalpro reports errors during the match phase.

```
tie input \A.top.dclk_one_ram \A.top.dclk_one
tie input \B.top.dclk_one   \B.top.dclk_one_ram
```

This next example is similar to the first example, except the tied ports are complements of each other.

```
tie_compl input \A.top.dclk_one_a \A.top.eclk_one_a
tie_compl input \B.top.dclk_one_a \B.top.eclk_one_a
```

The next example shows a situation where a port named **port1** exists in both the A-design and B-design. Four input ports named **port1_ofn1**, **port1_ofn2**, **port1_ofn3**, and **port1_ofn4** exist only in the B-design.

In the example, each of the four statements tie one of the four ports to **port1**. The intent is that this group of tied ports in the B-design will be matched with **port1** in the A-design.

Note that the statements maintain the required parallel structure; **port1** is on the left side in each statement. Also note that **port1** is the only port that exists in both designs. Putting **port1** on the left side in each statement makes it the “identifier” of the tied group, and enables FormalPro to correctly match **port1** in the A-design with the “tied” group of ports on the B-side.

```
tie input \B.top.port1 \B.top.port1_ofn1
tie input \B.top.port1 \B.top.port1_ofn2
tie input \B.top.port1 \B.top.port1_ofn3
tie input \B.top.port1 \B.top.port1_ofn4
```

The next example shows how tie an input port to an output port.

```
tie \A.top.in[0] \A.top.out[1] //connect out[0] to in[1]
```

Related Commands

For similar commands that apply to registers, see “duplicate and duplicate_compl” on page 223.

multiplierarchitecture

Scope: Constraint file

Specifies the architecture of the logic generated for a specific multiplier instance when compiling RTL for an RTL-to-gate equivalence check.

This can improve the solve efficiency of multipliers by enabling you to match the logic of multiplier generated from the RTL to the logic of the gate-side multiplier.

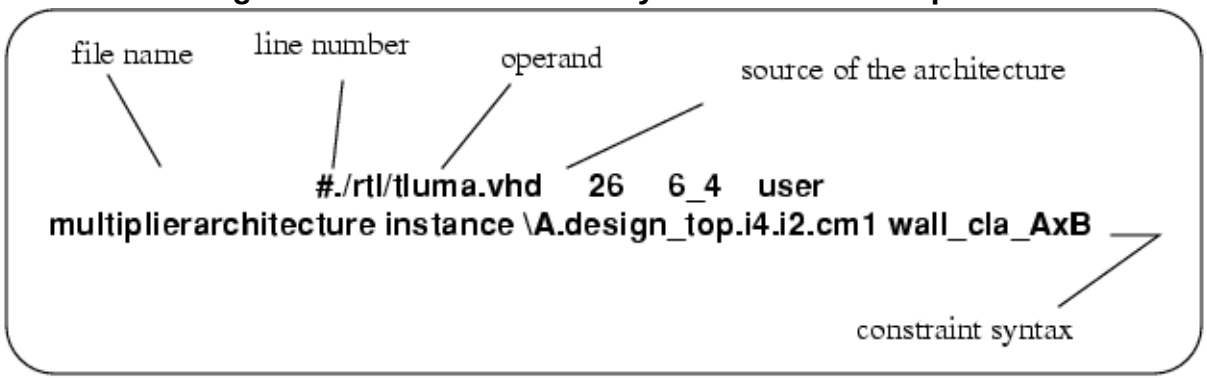
In the initial run the compiler determines which multiplier logic to use based on what it finds in the RTL source, and produces a Multiplier Architecture report.

The report lists every multiplier instance and shows the architecture of the logic generated by complier. It shows this using a syntactically correct multiplierarchitecture constraint that would generate the same results (see Figure 5-3)

If a given multiplier is taking too long to solve, you can try using a different architecture:

1. Find the multiplier instance in the report.
2. Copy the constraint from the report into the constraint file, then edit the syntax as desired.
3. Restart the run. The first time you change an multiplier architecture, restart at the compile phase. Thereafter when you add or revise a multiplierarchitecture constraint, recompilation is not necessary — you can restart at the match or constraint phase.

Figure 5-3. Format of an entry in the Multarch report.



The report is located here:

```
formalpro.cache/reports/multarch.report
```

Syntax

multiplierarchitecture instance instancePath arch[_adder][_swap]

Arguments

- instancePath — identifies the specific multiplier affected by this command.
- arch — compiles multipliers with a specified architecture. Options include:
  - wall — Wallace tree architecture.
  - csa — Carry-save Adder architecture.
  - str — Pipelined Wallace tree architecture.
  - nbw — Non-Booth Wallace architecture.
  - mcarch — architecture of Synopsys Module Compiler. There is no associated adder type.
  - csmult — Synopsys DesignWare multiplier. You must specify -strategy EXTREME if you use this argument.
  - pparch — Same as csmult (see above).
- adder — specifies the final adder architecture within the multiplier. Options include:
  - cla — Carry Look Ahead adder (default; if using csa)
  - rpl — Ripple adder
  - csel — Carry Select adder
  - csm — Conditional Sum Module adder (default; if using wall)
  - bk — Brent Kung adder (default; if using nbw)
- swap — specifies the swap order of the operands. Options include:
  - axb — compile the operands as specified in the RTL (default)
  - bxa — swap the order of the operands before compiling

Examples

The following example specifies that a multiplier instantiated in \A.top.mult should be compiled with the Non-booth Wallace architecture with a Carry-look-ahead adder with the default swap order.

```
multiplierarchitecture instance \A.top.mult.mult_5 nbw_cla_axb
```

See Also

For an overview of how multiplier architectures are determined, see “Specifying Multiplier Architectures” in the FormalPro User’s Manual

See also the "-multiplierArchitecture" command line switch.

make_pi and make_po

Scope: Constraint file

Splits combinational feedback loops (cycles).

If your design contains combinational feedback loops (cycles), FormalPro reports them in the cycles.report file in the reports directory of the FormalPro cache.

You can break these cycles for verification by using the make_pi and make_po commands. These commands split the feedback net into a primary input and a primary output.

To determine which net is causing the cycle, refer to the cycles.report file. You should notice a group of net names under the heading "Cycle number"; the nets listed here are involved in the cycle:

```
#
#Cycle number cc1 drives target tf161d state = Unsolved
#
MCP \A.top.i3.tja_it
MCP \A.top.i3.tja_it
```

The net listed is a good candidate for correcting the loop. Based on your knowledge of the design, you may know which net is causing the cycle and, therefore, do not need to analyze the report to determine the responsible net. Use this net name as the netName argument in the make_pi/make_po commands to determine if the combinational cycle is no longer reported by FormalPro.

Note

Use the *formalpro -restart match* command to restart a run after creating or altering a cycle breaking constraint. See -restart.

You can view a graphical representation of a combinational cycle within the FormalPro debug tool by using the showschematic command on a target that is classified as "fed by cycle".

Syntax

make_pi netName

make_po netName

Arguments

- netName

  Variable that must be a complete path of a net instance name.

Examples

For example, if the cycle report lists the net *\A.top.sub.status* twice in one cycle, and "MCP" (matched cut point) precedes both occurrences, you should add the following to the constraint file:

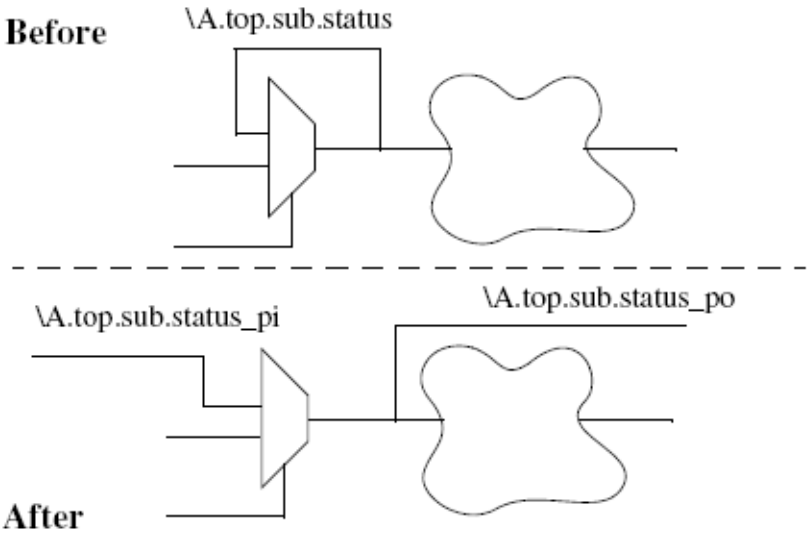
```
make_pi \A.top.sub.status
make_po \A.top.sub.status
```

You will also need to write similar commands for the corresponding net in the B design. For this example, assume that its name is similar to the name of the net in the A design.

```
make_pi \B.top.sub.status
make_po \B.top.sub.status
```

Figure 5-4 shows the results of applying the constraints to the A design. The results are similar for the B design.

Figure 5-4. Example: Cycle Breaking



Don't Care

You can instruct FormalPro on how to treat *don't care* situations within your designs.

Scenarios where you would possibly use these don't care constraints are:

- Performing hierarchical verification on a sub-module when you know that certain input conditions will never occur.
- Specifying onehot or onecold FSM encoding schemes.

- Specifying detailed information on how to translate an RTL design containing don’t care conditions.

Key terms used for this section:

- Group — refers to the group constraint and all of its arguments and variables.
- Signals — objects within the design, recognized by FormalPro, including input ports, registers, and nets.
- Condition — specifies vectors relevant to the signals listed in a group.

There are three types of constraint commands, all of which are dependent on each other.

- Specifying Groups — specifies a group of signals, based on which FormalPro could create a don’t care.
- Specifying Conditions — specifies the conditions, under which a target is defined as don’t care.
- Specifying Targets — allows you to instruct FormalPro to create a don’t care, dependent on the referenced Conditions and Groups.

To restart a run after altering a *don’t care* constraint, use the *-restart constraint* option. FormalPro determines whether it should restart at the compile or match stage, based on the contents of the constraint file.

Specifying Groups

Use the group command to specify a group of signals (inputs, registers, or modules), to which a set of conditions could apply.

Usage

```
group {inputs | registers} groupName
signal
[…]
endgroup

group groupName
{r | i | n} signal
[…]
endgroup

group module {A | B} moduleName groupName
moduleSignal
[…]
endgroup
```

Arguments

- inputs | registers

  An argument that specifies that the signals listed between the group and endgroup keywords are inputs or registers, respectively.

- groupName

  A variable that specifies a name assigned to the group of ports or registers listed between the group and endgroup keywords. The dc, onehot, and onecold commands will reference this name.

- signal

  A variable that must be a fully specified path name to a scalar port or scalar register. This variable can also be a register alias, as shown in the Match tool. FormalPro considers the first signal listed as the LSB, and the last signal as the MSB. You can only specify one signal per line within a group constraint.

- r | i | n

  Arguments that allow you to specify a mixture of signals in one group command. Each argument must be followed by only one signal, and only one argument/variable pair is allowed per line.

  - {r | register} — specifies that the corresponding signal is a register.
  - {i | input} — specifies that the corresponding signal is an input port.
  - {n | net} — specifies that the corresponding signal is a net.

- module

  An argument that specifies that the moduleSignals listed between the group and endgroup keywords are inputs, nets, or registers within moduleName.

- A | B

  An argument that specifies that moduleName is in a particular design.

- moduleName

  A variable that must be a name of a module within the specified design.

- moduleSignal

  A variable that must be a path name to a scalar port or scalar register within the specified moduleName. FormalPro considers the first moduleSignal listed as the LSB. You can only specify one moduleSignal per line within a group constraint.

Specifying Conditions

These commands allow you to specify the conditions that must exist for a given group of signals, for FormalPro to create a don’t care.

Three constraint commands apply to this category:

- dc — specifies a collection of conditions that could apply to a specific groupName (used in the group constraint).
- onehot — specifies a groupName as a one hot.
- onecold — specifies a groupName as a one cold.

Syntax

dc dontCareName groupNamecondition[...]enddc

onehot dontCareName groupName

onecold dontCareName groupName

Arguments

- dontCareName

  A variable that specifies a name assigned to the group of conditions listed between the dc and enddc keywords, or assigned to onehot or onecold constraints. The applydc and applydc_compl commands will reference this name. When you use the onehot constraint, you should reference it’s dontCareName in an applydc_compl command.

- groupName

  A variable that references the name of a specific group, to which the condition is applied.

- condition

  A collection of binary or hex assignments. The allowable formats, as well as examples, are as follows:

  - binary format — 000000
  - binary format with don’t care notation — 0-010-
  - hex format — 042F
  - hex format with don’t care notation — 0-2-

  You can only specify dash characters (-) for don’t care conditions in both binary and hex format assignments. You can specify more than one condition for a dc constraint, but can only specify one condition per line. FormalPro reads the condition with the LSB on the left. The length of the condition, should match the number of signals in groupName.

Specifying Targets

These commands specify which target is to be defined as a don't care based on the information in a specific dontCareName.

There are two constraint commands for specifying targets:

- applydc — applies a don't care to a target when a valid condition is listed in the corresponding dc constraint.
- applydc_compl — applies a don't care to a target when a valid condition is not listed in the corresponding dc constraint.

FormalPro applies a don't care to any target that depends on any signal in a group identified by dontCareName. This constraint, when specified without the register variable, could cause FormalPro to unintentionally apply don't care conditions to some targets and mask real differences.

If the target is a register, FormalPro applies a don't care to the data and q-output pins (only if you specified -diffOnQ), by default. You can specify other pins to receive a don't care through the use of optional arguments. If you specify a terminal that has no path to any signal of the referenced group, FormalPro will still apply a don't care to that terminal.

FormalPro produces a report listing all targets to which don't care conditions are applied: dontCare.index in the outputFiles directory of the FormalPro cache.

When you need to specify many applydc or applydc_compl commands, you should refer to the section "Constraint and Match File Scripts".

Syntax

applydc dontCareName [register]
[-r] [-s] [-d] [-c] [-q]

applydc_compl dontCareName
[register] [-r] [-s] [-d] [-c] [-q]

Arguments

- dontCareName

  A variable that references the name of a specific dc condition group, against which the target is checked.

- register

  An optional variable that must be a fully-specified pathname to a scalar register. This command allows you to explicitly specify a register for don't care treatment. If the conditions and signals referenced by this command apply to the specified register, FormalPro, by default, applies a don't care to the data and q-output pins of the register.

- -r -s -d -c -q

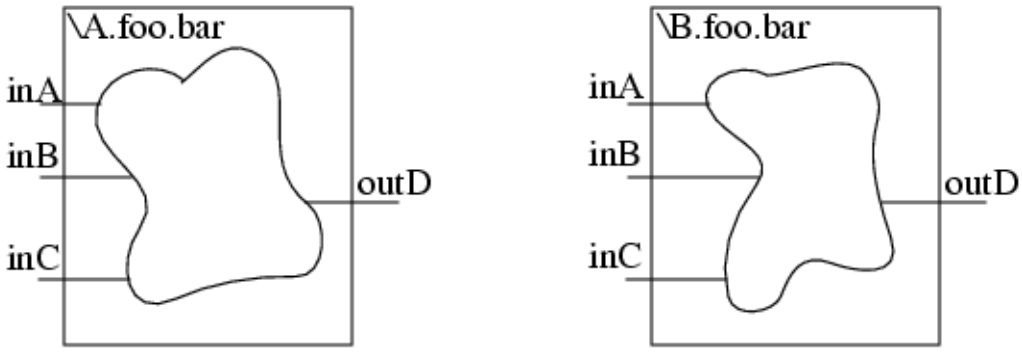
  These optional arguments instruct FormalPro to apply don’t cares to specific pins of registers affected by the referenced conditions and signals. If you do not specify any of these switches, FormalPro, by default, sets the -d and -q switches.

-r	reset pin	-s	set pin
-d	data pin	-c	clock pin
-q	q-output pin		

Examples

Assume that in the modules shown in Figure 5-5, \A.foo.bar[outD] is equivalent to \ B.foo.bar[outD] only when the inputs to the modules are all “1”, all “0”, or inA and inB are “0” and inC is “1”:

Figure 5-5. Advanced Constraint Example



You can instruct FormalPro to only verify the equivalence in these cases by adding the following commands to your Constraint file:

```
# Specify a group of signals in design A that must be set for
# the scenario to occur. LSB top... MSB bottom.

group in_ABC
   i \A.foo.bar.inA
   i \A.foo.bar.inB
   i \A.foo.bar.inC
endgroup

# Specify the conditions that apply to the signals in a
# specific ‘group’.
# (left column represents ‘inC’ and right column ‘inA’).

dc valid_states in_ABC
   111
   000
   100
enddc

applydc_compl valid_states
```

Configuration Files

A configuration file is a special type of constraint file processed during RTL compilation that modifies the internal database before the matching phase. Configuration files are specified with the **-configFile** command option.

For more information, see -configFile.

To apply any changes to a configuration file, the session must be restarted from the “Compile” phase.

The following commands can be used in a configuration file:

encode

Scope: Configuration file

Specifies encoding of state machines. Ensures that state machines in A-side RTL conform to the encoding used in the B-side netlist.

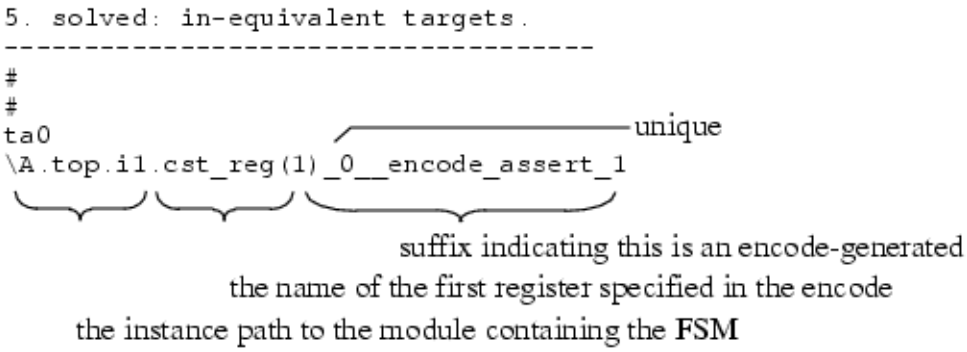
When an encoding is specified that does not specify all possible states, FormalPro checks the A design-side (RTL) to ensure that the unspecified states cannot occur. If one of the states can occur, an assertion failure is reported. Shown below is an encode command generated by transFVI based on output from a Precision synthesis run. Note that only three of the four possible states are defined. Precision is declaring that the fourth state (10) cannot occur and that it will base optimizations on that assumption.

```
encode  -module fsm_encoding -reference cst_reg(1) cst_reg(0) -translated
cst_reg(1) cst_reg(0) {
    00 : 00
    01 : 01
    11 : 11
```

Given this encode command, FormalPro checks to see if the invalid state “10” could occur on the A design-side. (The pre- and post-encoded states are the same in this example, but they may differ.)

If an assertion failure is reported in the log file, go to the Detailed Comparison report to determine if it is encode-generated. If it is, the entry in the report gives you the information you need debug the problem: the path to the module in the RTL source where the FSM is defined and a register name that will enable you to find the encode command in the configuration file.

Figure 5-6. Assertion Failure in the Detailed Comparison Report.



To resolve an encode-generated assertion failure:

- Find the encode statement by searching the configuration file for the register named in the Detailed Comparison report and review the encode statement; fill-in any missing codes.
- Review the synthesis tool settings and configuration.
- Review the RTL source to confirm that the RTL input to FormalPro is the same RTL input to the synthesis tool.

If you decide everything is okay as is, you can disable the encode-generated assertion failure by adding -noassert to the encode command in the configuration file.

Syntax

encode -noassert -module *moduleName* -reference *registerList* -translated *registerList* {*encodingMap*}

Note

Line breaks are allowed only between the curly braces {}, to delineate items in the encoding map. The line continuation backslash (\ ) is not supported.

Arguments

- -noassert

  (optional) disable assertion test for missing states in encoding table.

- -module *moduleName*

  were *moduleName* is the A-side module containing the state machine to be encoded.

  VHDL module names can be specified in various formats:

  - *libName.entName(archName)*
  - *entName(archName)*

- *entName*
- *libName-entName-archName*
- *libName-entName-generics-archName*

The following example specifies the default library *work*, entity *xdr_tmg*, and architecture *rtl*:

```
-module work-xdr_tmg-rtl
```

If you have trouble with the correct specification of a VHDL module name, look it up in the files in the FormalPro cache after compilation (.cache/internal/T/A/NET).

- -reference *registerList*

  where *registerList* is a space delimited list of the registers in A-side RTL-level state machine.

- -translated *registerList*

  where *registerList* is a space delimited list of the registers in B-side, gate-level state machine.

- {*encodingMap*}

  where encoding map is a space delimited list of A-side/B-side mapping of reachable states.

Example

Note

Line breaks are allowed only between the curly braces {}. The example below may display line breaks due to the page-width limits and are not valid in an actual command.

In the following example, the RTL specifies a six-bit state machine and in the B-side netlist it is optimized to 2-bits:

```
encode -module dflop -reference reg_mystate(5) reg_mystate(4)
reg_mystate(3) reg_mystate(2) reg_mystate(1) reg_mystate(0) -translated
reg_mystate(1) reg_mystate(0) {
010111 : 00
101011 : 01
001101 : 10}
```

partial_sum_checker

Scope: Configuration file

Specifies a module with partial-sum outputs. Adds sum-checking logic to verify partial sum outputs.

The two partial sum output ports and the name of the SystemVerilog/Verilog module or VHDL entity/architecture must be supplied.

The -partialSumCheck option must also be specified for the FormalPro run. Module names that contain "DW02_multp" or "DW02_tree" are handled automatically when -partialSumCheck is used.

Syntax

partial_sum_checker {-reference
| -translated} <*module_entity/architecture*>
<*outName0*>
<*outName1*>

Arguments

- -reference

  Specifies a module in the A design.

- -translated

  Specifies a module in the B design.

- *module* or *entity/architecture*

  Specifies the name of the Verilog module or VHDL entity/architecture to add the sum checker logic to. Verilog escaped names and VHDL extended identifier syntax is supported.

  VHDL Entity/architectures should have the form: <library>.<entity>(<arch>)

- *outName0*

  Specifies the first output port to sum.

- *outName1*

  Specifies the second output port to sum.

Examples

The following example adds sum checking logic to outputs *p0* and *p1* of the *share.multp_6x6* module in design A and to the *multp_6x6_param* module in Design B.

```
partial_sum_checker -ref share.multp_6x6 p0 p1
partial_sum_checker -tra multp_6x6_param p0 p1
```

port_direction

Scope: Configuration file

Specifies the direction of a port for RTL compilation.

It applies to top level and blackbox ports, and typically switches a bidirectional port to an input or an output port.

Syntax

port_direction {-reference
| -translated } {Input | Output } <*portName*>

Arguments

- -reference

  Specifies a module in the A design.

- -translated

  Specifies a module in the B design.

- Input | Output

  The direction of the port.

- *portname*

  The name of the port.

Examples

The flowing example specifies the direction of two ports in the A design: *data_out1* as an output and *data_input* as an input:

```
port_direction -reference Output data_out1
port_direction -reference Input data_input
```

Appendix A
FormalPro Library Compiler

You can use the FormalPro Library Compiler (fplibcomp) to compare two libraries or precompile libraries to a Verilog format for use with FormalPro.

You can use *fplibcomp* to compare libraries in the following formats:

- Verilog
- FastScan ATPG
- Synopsys Liberty

Comparing Libraries

The FormalPro Library Compiler allows you to verify the equivalency of two different libraries. You may want to compare libraries if you suspect a problem between two different revisions of the same library or if you want to make sure that the functionality described in two different formats of the library is equivalent. The following three-step procedure uses the FormalPro Library Compiler and FormalPro to perform the comparison.

Procedure

1. Use fplibcomp to precompile one of the libraries with the command line argument, -netlist netfilename. This generates a Verilog netlist that contains a single module instantiating all the modules of the library.
2. Perform Step 1 on the other library.
3. Use formalpro to compare the netlists created during the first two steps.

Examples

For example, to compare a Verilog library in a file called *libVlog.v* against a library in a Mentor Graphics FastScan ATPG format that is in file *libAtpg.v*, enter the following commands:

```
fplibcomp -vlib libVlog.v \
            -outputFile pre_comp_libVlog.v \
            -netlist netlist_libVlog.v \
            -cache fplibcomp_A.cache

fplibcomp -alib libAtpg.atpg \
            -outputFile pre_comp_libAtpg.v \
            -netlist netlist_libAtpg.v \
            -cache fplibcomp_B.cache

formalpro -a netlist_libVlog.v -v pre_comp_libVlog.v \
            -b netlist_libAtpg.v -v pre_comp_libAtpg.v
```

FormalPro recognizes that each design (netlist_libVlog.v and netlist_libAtpb.v) contains a single module called MGC_GENERATED_NETLIST and produces a report called library_comparison.report in the report directory. This report provides an overview of the equivalency of the two libraries, including modules that are:

- Equivalent or different
- Declared as blackboxes
- Found in only one design
- Unsolved

You can also read the other reports from the formalpro.cache to analyze the comparison of the libraries. For example, the detailed comparison report indicates the comparison points that make a library cell in design A different from the same cell in design B. The unmatched and detailed comparison reports list the reasons why some modules could not be solved. However, to find why a particular library module became a black box, use the reports produced by fplibcomp in the first two steps of the procedure.

Precompiling Libraries

To precompile libraries, use the following command line syntax (assuming Verilog libraries).

fplibcomp -vlib originalDir -outputdir newDir

where *originalDir* is the location of the Verilog libraries that you want to precompile, and *newDir* is where fplibcomp should create the precompiled libraries.

When you use a precompiled library with the *formalpro* executable, be sure to use the -y switch if you created a directory (-ouputDir) or the -v switch if you created a file (-outputFile).

fplibcomp

Executable that compiles design libraries for FormalPro.

Usage

fplibcomp ***library_format*** {**-outputDir *dir Pathname*** | **-outputFile *filePathname***}
[-netlist *netlistName*] [*globalOptions*]

library_format
{-alib[F] | {-slib[F]} | [-vlib[F]} {***file*** | ***fileList***}

globalOptions
[-cache *cacheDir*] [-mod *moduleName*] [-f *commandFile*] [-blackboxFile *fileName*]
[-stopOnBlackBox] [-stopOnMissing] [-libConfigFile *configFileName*]
[-[no]LibertyPGpins][-QQbarMerge] [-QQbarSetResetMerge] [-floatConnectValue]
[-masterSlaveMerge] [-redundantRegMerge]
[{-suffixVlogLib | -suffixDftLib | -suffixSynLib} *extensionList*]
[-hdlin_pragma <*directiveLabel*> [:<*directiveLabel*>] ...] [+incdir+*include_dir* …]
[+define+*definition=value* …] [-blackboxMemories] [-rtlMemoryLimit [*integer*]]

log_file_control
[-log *logFileName*] [-logLevel [mini | compact | full]] [-overWrite]

fplibcomp {-help [blackbox] | -version}

Description

There are four parts to the command syntax as follows:

- *library_format* specifies the input technology library (for example, files or directory and type).
- **-outputFile** or **-outputDir** specifies the file or directory where compiled libraries are output.
- **-netlist** generates a netlist that instantiates all the library cells that can be compared to another netlist library using **formalpro**. For more information, see Comparing Libraries.
- *globalOptions* consists of the optional arguments that allow you to customize the compilation session. Global options can be specified anywhere on the command line.

Arguments

- -alib *library*

  Specifies the file or directory location of FastScan ATPG technology libraries where *library* is either a single library file or a directory containing multiple library files. Only one *library* arguments can be specified per switch.

  All files with an .lib or .atpglib suffix contained in a specified directory are loaded. All pathnames are relative to the working directory.

- -alibF *fileList*

  All files with an .lib or .atpglib suffix contained in a specified directory are loaded. All pathnames are relative to the working directory.

  The format of the *fileList* file is shown in the following example:

```
# commented line
./lib/atpg_library_1.lib          # a single ATPG library file
./lib                             # a directory location containing
                                  # ATPG library files
./lib/atpg_library_*.lib          # wildcards are allowed
```

- -blackboxFile *fileName*

  Specifies the location of a black box file containing user-created black box definitions.You can also use the alias -bbfile. Non-literal pathnames are relative to the current directory. The syntax for the black box file is:

```
# commented line
blackbox  <moduleName>
```

  The FormalPro Library Compiler creates an empty module when you specify a module or cell as a black box. This functionality is most useful for RAM cells not normally verified with FormalPro.

- -blackboxMemories

  Instructs the FormalPro Library Compiler to black box memory modules larger than 1K.

  - -blackboxMemories — enables this functionality (default).
  - -noblackboxMemories — disables this functionality

  When fplibcomp encounters a module that appears to be a large memory, it black boxes the module, issues an error notifying you of the action, and continues on with the run.

  This functionality prevents fplibcomp from using CPU time and memory on the compilation of memory modules.

  To compile and verify the memory modules, specify the -noblackboxmemories switch.

- -cache *cacheDir*

  Specifies a path and directory name for the FormalPro Library Compiler cache.

  A new cache directory is created if none exists. If a cache already exists at the specified location, it is overwritten. The default location for the FormalPro cache is ./ fplibcomp.cache.

- +define*definition=value*

  Specifies a Verilog definition in one of your library files.

  - *value* — a value assigned to the definition (optional).
  - *definition*— the name of the definition.

You should use this switch if your Verilog RTL files contain ‘ifdef statements and you want the FormalPro Library Compiler to compile the code within the statement.

- -f *commandFile*

  Loads a command file containing additional command line switches.

  - *commandFile* — the name of the command file.

  Loading a command file allows you to specify a text file containing frequently used command line switches. The syntax for the commandFile is as follows:

  ```
  # commented line
  <switch> ... \
  <switch> ...
  ```

  Be sure that no uncommented lines have newline characters; otherwise, the command line will not run properly. Always use a backslash (\) when specifying switches across multiple lines.

- -floatConnectValue {0 | 1 | X | Z}

  Specifies the driving value of all floating nets in libraries during compilation. Options include:

  - 0 — floating nets are connected to the value 0.
  - 1 — floating nets are connected to the value 1.
  - X — floating nets are connected to X.
  - Z — floating nets remain unconnected (default).

  The FormalPro Library Compiler applies this switch to floating nets in your libraries during pre-compilation.

- -help

  Displays help information on the usage of the fplibcomp.

  - -help [blackbox] — when you specify this switch with no argument, you receive a help file for the formalpro command.

    blackbox — displays help on creating black box files.

- -hdlin_pragma

  Specifies directive labels used in a design.

  - <directiveLabel>:[<directiveLabel>]...

  "ambit:exemplar:synopsys:pragma" is the default.

- +incdir

  Specifies the location of a directory referenced by a Verilog include statement.

  - +incdir+include_dir

include_dir — specifies the directory that contains your included files.

You should use this switch if your Verilog RTL files contain ‘include statements and you want the FormalPro Library Compiler to compile the code within the referenced file.

Your ‘include statements generally reference a file, but for this switch, you need to specify the directory that contains the file, not the file itself.

When you have multiple ‘include statements that point to different directories, you must specify +incdir+ for each directory.

- -libConfigFile

  Specifies rules for replacing Verilog library cells.

  - -libConfigFile *configFileName* — You can also specify the alias -lcf.

    configFileName — specifies the location of the text file containing the replacement rules.

  In some cases, Verilog technology libraries may contain behavioral or timing-check constructs that you need to remodel for FormalPro to compile them. Once you have obtained these remodeled libraries, you need to map the original libraries to the new libraries using a library configuration file.

  The format of the library configuration file (*configFileName*) consists of one entry per line, where FormalPro ignores blank lines and comment lines prefixed by the number sign (#). Each entry line has the following syntax:

  ```
  <cellToReplace> : <replacementCell> (<verilogPortList>);
  ```

  where cellToReplace and replacementCell are cell names, separated by a colon (:), and verliogPortList is a comma-separated list of port names enclosed in parentheses. You must end the entry line with a semicolon (;), as shown in the following example:

  ```
  # commented line
  cellA : cellA_new (q, a, b);
  ```

  You must load both the original library and the remodeled library with the library specification switches (-alib, -vlib, -ylib, or -slib).

- -[no]LibertyPGpins

  Either include or exclude power and ground pins declared on a CELL in a Liberty library.

  -LibertyPGpins is the default.

- -log

  Renames the FormalPro Library Compiler log files.

  - -log *logFileName*

    logFileName — a string used for the new log file name.

  This switch renames the file fplibcomp.log to <logFileName>.log.

- -logLevel

  Controls the amount of information written to the FormalPro Library Compiler log file and stdout.

  - -logLevel [mini | compact | full]

    mini —writes only the final status.

    compact —writes summary information for each stage (default).

    full — writes all detailed information for each stage.

  FormalPro always generates all three log files for each run. You can find them in the following FormalPro cache location:

  ```
  fplibcomp.cache/logs/
     fplibcomp_mini.log
     fplibcomp_compact.log
     fplibcomp_full.log
  ```

  These filenames may be altered, depending on your setting for the -log switch.

- -masterSlaveMerge

  Compiles library files containing two registers in a master/slave configuration so that the register is represented as a single DFF.

  - -masterSlaveMerge — enables this functionality (default).
  - -nomasterSlaveMerge — disables this functionality.

  Use this functionality so that the number of registers and their appearance are similar to your post-synthesis design. In some cases, the technology libraries used during synthesis could increase the register count.

- -mod

  When using -netlist, this switch limits the library compilation to the module specified. You can use this switch only when you specify -netlist.

  - -mod *moduleName*

    moduleName — name of the module to compile.

- -netlist

  Generates a netlist that instantiates every module of the specified library.

  - -netlist *netlistName*

    netlistName — location and name of the netlist to be created.

  Use this switch when comparing two versions of a library using FormalPro.

- -outputDir

  Specifies a directory location for the precompiled libraries.

  - -outputDir *dirName* — creates a directory containing one file for each precompiled library module. Each file has the name <moduleName>.v. You can also use the alias -od.

    dirName — location and name of the directory to be created.

  The FormalPro Library Compiler generates the precompiled libraries in a simple Verilog format that FormalPro is able to parse and use in a more efficient way.

- -outputFile

  Specifies a file name and location for the precompiled libraries.

  - -outputFile fileName — creates a single file containing a precompiled version of every library file. You can also use the alias -of.

    fileName — location and name of the file to be created.

  The FormalPro Library Compiler generates the precompiled libraries in a simple Verilog format that FormalPro is able to parse and use in a more efficient way.

- -overWrite

  Overwrites existing generated files and directories.

- -QQbarMerge

  Compiles library files containing two registers representing Q and Qbar ports so that the registers are represented as a single DFF.

  - -QQbarMerge — enables this functionality (default).
  - -noQQbarMerge — disables this functionality.

  Use this functionality so that the number of registers and their appearance are similar to your post-synthesis design. In some cases, the technology libraries used during synthesis could increase the register count.

- -QQbarSetResetMerge

  Compiles library files containing set- or reset-dominant registers defined with two UDPs so that the registers are represented as a single DFF.

  - -QQbarSetResetMerge — enables this functionality (default).
  - -noQQbarSetResetMerge — disables this functionality.

  Use this functionality so that the number of registers and their appearance are similar to your post-synthesis design. In some cases, the technology libraries used during synthesis could increase the register count.

- -redundantRegMerge

  Compiles library files containing two UDPs that produce equal output so that the registers are represented as a single DFF.

  - -redundantRegMerge — enables this functionality (default).

  - -noredundantRegMerge — disables this functionality.

Use this functionality so that the number of registers and their appearance are similar to your post-synthesis design. In some cases, the technology libraries used during synthesis could increase the register count.

- -rtlMemoryLimit

  Specifies an upper limit for the size of a memory module to be compiled.

  - -rtlMemoryLimit [integer]

    integer — The default value of integer is 1024.

  This switch alters the default size for the -blackBoxMemories switch, which automatically black boxes large memory modules. Memories that are larger than the integer value you specify are subject to the -blackboxMemories switch.

- -slib

  Specifies technology library files in Synopsys Liberty format.

  - -slib *library* —specifies the file or directory location of Synopsys Liberty technology libraries.

    library — a single library file or a directory containing multiple library files. One library argument per switch.

  - -slibF *fileList* — specifies the location of a file listing the location of Synopsys Liberty technology libraries.

    fileList — the location of the file. One fileList argument per switch.

  When you specify a directory as an argument to -slib, or within the -slibF file list, the FormalPro Library Compiler recursively expands the directory and loads all files with .lib or .synlib suffixes. You can specify these switches any number of times on one command line.

  The format of the file specified by fileList is shown in the following example:

```
# commented line
./lib/syn_library_1.lib          # a single Synopsys library file
./lib                            # a directory location containing
                                 # Synopsys library files
./lib/syn_library_*.lib          # wildcards are allowed
```

- -stopOnBlackBox

  Instructs the FormalPro Library Compiler to end a run if it black boxes a module that was not user-specified.

  - -stopOnBlackBox — enables this functionality.

  - -nostopOnBlackBox — disables this functionality (default).

- -stopOnMissing

  Instructs the FormalPro Library Compiler to end a run if it encounters a missing library cell.

  - -stopOnMissing — enables this functionality. You can also use the alias -somg.
  - -nostopOnMissing — disables this functionality (default). You can also use the alias -nosomg.

- -suffixDftLib

  Specifies the suffix styles used for FastScan ATPG files.

  extensionList — a colon-separated list of extensions.

  Default extensions: .atpglib and .lib.

- -suffixSynLib

  Specifies the suffix styles used for Synopsis Liberty library files.

  extensionList — a colon-separated list of extensions.

  Default extensions: .synlib and .lib.

- -suffixVlogLib

  Specifies the suffix styles used for Verilog library files.

  extensionList — a colon-separated list of extensions.

  Default extensions: .v and .V.

- -version

  Displays version information for the FormalPro Library Compiler.

  - -version

- -vlib

  Specifies technology library files in Verilog format.

  - -vlib *library* —specifies the file or directory location of Verilog technology libraries.

    library — a single library file or a directory containing multiple library files. One library argument per switch.

  - -vlibF *fileList* — specifies the location of a file listing the location of Verilog technology libraries.

    fileList — the location of the file. One fileList argument per switch.

  When you specify a directory as an argument to -vlib, or within the -vlibF file list, the FormalPro Library Compiler recursively expands the directory and loads all files with .v or .V suffixes. You can specify these switches any number of times on one command line.

  The format of the file specified by fileList is shown in the following example:

```
# commented line
./lib/ver_library_1.v            # a single Verilog library file
./lib                            # a directory location containing
                                 # Verilog library files
./lib/ver_library_*.v            # wildcards are allowed
```

Examples

The following example pre-compiles an ASIC library in the Mentor Graphics ATPG format and puts the restricted gate-level Verilog in the file *resultFile.v*.

```
fplibcomp -alib asic_lib.lib \
      -outputFile resultFile.v
```

This command pre-compiles a Verilog library and puts the results in a directory called *myPreCompiledVersion*.

```
fplibcomp -vlib asic_lib.v -outputDir ./myPreCompiledVersion
```

Simple Verilog Format

The FormalPro Library Compiler produces the precompiled libraries in a “simple” Verilog format.

The simple format is as follows:

- Supports net descriptions using wire and supply (scalar and vector).
- Supports modules and ports (in, out, inout - scalars and vector)
- Supports Verilog built-in primitives, except transistors
- Supports integer constants, except all forms of “X” values
- Supports pre-processor instructions and comments
- Does not support instantiations of sub-modules
- Does not support assignments
- Does not support control flow constructs (always, initial, and so on) or expressions
- Does not support timing (including specify blocks) or drive strength
- Does not support UDPs

To provide the additional functionality to FormalPro, two primitives were added to Verilog. You can use these primitives to look at the description of some cells when debugging the circuit with the schematic to see which gates are associated with a particular library cell. You could also use these primitives to modify the precompiled format before using it in FormalPro. The primitives are:

```
primitive __mgc_fv_dff
primitive __mgc_fv_dlatch
```

One represents an edge-sensitive DFF and the other a level-sensitive latch. Their behavior is represented in the following descriptions using two Verilog User-Defined Primitives (UDP). You can use these descriptions as inputs to a Verilog simulator, such as ModelSim.

```
primitive __mgc_fv_dff (Q, D, CLK, SET, RESET);
    output Q;
    input  D, CLK, SET, RESET;
    reg    Q;
    // Positive edge triggered D flip-flop with active high
    // asynchronous set and reset.
    //  -> nor set nor reset dominates. Explicitly emits an X
    //      in case of conflicts.
    initial Q = 1'bx;  // make sure we start with an X...



    table
      // D  CLK   SET   RESET    Qt  Qt+1

         1  (01)  0     0     :  ?  : 1;// clocked data
         0  (01)  0     0     :  ?  : 0;  // clocked data

         ?  ?     *     0     :  1  : 1;  // pessimism
         ?  ?     0     *     :  0  : 0;  // pessimism

         ?  ?     1     0     :  ?  : 1;  // asynchronous set
         ?  ?     0     1     :  ?  : 0;  // asynchronous reset
         ?  ?     1     1     :  ?  : X;  // conflict set/reset -
                                                                    /
/ make it explicit

         ?  (?0)  ?     ?     :  ?  : -;  // ignore falling clock
         0  (?x)  ?     ?     :  0  : -;  // retain state when D==Qt
         1  (?x)  ?     ?     :  1  : -;  // retain state when D==Qt
         *  1     ?     ?     :  ?  : -;  // ignore data edges
         *  0     ?     ?     :  ?  : -;  // ignore data edges
    endtable
endprimitive

primitive __mgc_fv_dlatch (Q, D, EN, SET, RESET);
    output Q;
    input  D, EN, SET, RESET;
    reg    Q;
    // Positive level sensitive D latch with active high
    // asynchronous set and reset.
    //  -> nor set nor reset dominates.  Explicitly emits an X
    //      in case of conflicts.

         initial Q = 1'bx;  // make sure we start with an X...

         table
      // D  EN    SET   RESET    Qt  Qt+1
         1  1     0     0     :  ?  : 1;  // enable data
         0  1     0     0     :  ?  : 0;  // enable data

         ?  0     0     0     :  ?  : -;  // pessimism

         ?  ?     1     0     :  ?  : 1;  // asynchronous set
         ?  ?     0     1     :  ?  : 0;  // asynchronous reset
         ?  ?     1     1     :  ?  : X;  // conflict set/reset -
                                                                    /
```

```
/ make it explicit
    endtable
endprimitive
```

Appendix B
Using EDIF Design Files

FormalPro accepts EDIF files as the design root for a gate-level design. FormalPro does not support the instantiation of an EDIF design within a top-level Verilog or VHDL design. The cells in “external” libraries in EDIF are considered primitives, and FormalPro searches the specified Verilog, VHDL or ATPG libraries for their models. If the model is not found in those libraries, it is black-boxed.

For more information on specifying input files, see “designFile” on page 43.

For more information about how FormalPro compiles EDIF designs, see “Compiled EDIF Designs” on page 268.

For best results, EDIF files must be optimized for FormalPro. Optimizing files includes assigning power or ground to nets and ports and specifying the EDIF suffix style used in your design. For more information, see the following sections:

Specifying Nets and Ports as Power or Ground

For designs in EDIF format, you must explicitly specify which ports and nets are tied to power and ground. FormalPro uses this information to produce a correct circuit description and for constant propagation.

Use the following switches to assign logical values to ports or nets that are tied to power or ground. All of these switches are design-specific and apply only to scalar ports and nets.

VDDport

Assigns the value “1” to the specified port(s).

```
-vddport <portName>[:<portName>]
```

portName — Specify multiple names as a colon-separated list.

GNDport

Assigns the value “0” to the specified port(s).

```
-gndport <portName>[:<portName>]
```

portName — Specify multiple names as a colon-separated list.

VDDnet

Assigns the value “1” to the specified net(s).

```
-vddnet <netName>[:<netName>]
```

netName — Specify multiple names as a colon-separated list.

GNDnet

Assigns the value “0” to the specified net(s).

```
-gndnet <netName>[:<netName>]
```

netName — Specify multiple names as a colon-separated list.

Specifying Design File Suffixes

If your design uses variations on the EDIF file extension that differ from the default extensions, you must specify the extensions used in the design. Specifying the suffix styles allows FormalPro to recognize them as EDIF.

By default, FormalPro recognizes the following file extensions: .edf, .edif, .EDF, .EDIF, .EDN, .edn

Use the following switch to specify the extensions used in your design if your extensions differ from the default settings.

suffixEDIF

Specifies EDIF extensions used in the design.

-suffixEDIF <extensionList>

where extensionList is a colon-separated list of extensions. When you specify an extension or group of extensions, they override the default settings. Therefore, you must include all possible extensions in your argument.

The default extensions are: .edf, .edif, .EDF, .EDIF, .EDN, .edn

Compiled EDIF Designs

FormalPro compiles EDIF design with limitations.

Compiled EDIF designs are compiled as follows:

- Treats all cells and views as if they are from the same library.
- Handles only the first view for each cell.
- Handles identifiers as case-sensitive.
- No limitation to the length of an identifier.
- Converts spaces of renamed identifiers to underscores.
- Does not support:
  - viewRef keyword
  - keywordAlias construct
  - keywordDefine construct
  - extended strings
  - instance arrays
  - subnets

Special Processing Rules

EDIF files have a field that specifies the source program that generated the file.

FormalPro uses this information to do some special processing of data in the EDIF file as follows:

- Precision-generated EDIF— The INIT property values associated with an instance of a cell are interpreted as 32-bit hexadecimal numbers to allow for programming of LUT cells.
- Synplify-generated EDIF — All but one of the "library" sections of this file are interpreted as being external, which means that they are assumed to have no netlist structure in EDIF and that their logic models are contained in an external Verilog, VHDL, or ATPG library. Only the "work" library is interpreted as having netlist structure in all its cell definitions.

Appendix C
FormalPro Utilities

FormalPro provides a collection of special-purpose utilities from the *fp_utility* command. These utilities perform a variety of useful functions such as encrypting/translating design files, resolving match issues, and exploring the results of equivalence checking.

fp_utility

The utilities are run via the *fp_utility* command in the following manner.

```
$ fp_utility <utility_name> <utility_args>
```

Utility key words: pipeline, retime, transparent, duplicate, unmatched, cross reference, analyze, FPGA routing, script, target report, solve loop, match loop, VSDC.

Most of the utilities provide a -help argument that displays additional usage information. For more information, email *fv_support@mentor.com*.

You can copy these utilities from the following location and customize them for your application:

```
$FORMALPRO_HOME/pkgs/fv/utils
```

Enter the *fp_utility* command from the Linux command line or via **Tool > Utility Shell** in the FormalPro GUI to display a list of the utilities similar to:

```
$ fp_utility
FormalPro Utilities 2.14
Usage:
     VSDC -help : translates VSDC[Synopsys guide files] and Oasys RT
                  formats
     CTC  -help : translates Cadence RC and Conformal command/guide file
                  and pragma conversion
     Xref -help : make matches from unmatched-Xref file conversion
     Goodmatches -help : make match + ignore files from equvalent results.
                         Usage:Goodmatches cache_name
     Unmatched -help : make unmatched-2-constraint extraction from
                       unmatched_objects.report file
     Make_duplicateB : make_duplicateB [cache] [-help]. Makes a
                        duplicateB.constraint file from cache if typical
                        SNPS _rep[i] is used. Edit script to customize

...
```

Appendix D
Supported VHDL2008 Constructs

This section describes key VHDL 2008 language features supported by FormalPro.

Conditional and Selected Sequential Assignments

VHDL 2008 allows conditional and selected signal assignment statements in concurrent (signals only) and sequential (signals and variables) modes.

Prior to VHDL-2008, this was limited to conditional and selected signal assignments in concurrent context only. A conditional statement (when?) carries the same semantics as an if-else-if statement. A select statement (select?) carries the same semantics as a case (case?) statement. By extending support for these conditional and select statements for variables and signals and also in sequential mode, the semantics of the language are made consistent. Thus, the choice list in a selected expression can even contain don't-cares (-) when the select? usage is done. The following example describes this clearly.

Example

VHDL 2002	VHDL 2008

`Process (d, reset)`
`Begin`
`  If reset = '1' then`
`  Q <= '0'`
`  Else`
`  Q <= d;`
`  End if;`
`End Process;`
	`Process (reset, d)`
	`Begin`
	`  Q <= '0' when reset else d;`
	`End process;`

`Process (val1, val2, val3, val4, sel)`
`Begin`
`  Case sel is`
`  When "00" => q <= val1;`
`  When "01" => q <= val2;`
`  When "10" => q <= val3;`
`  When "11" => q <= val4;`
`  End case;`
`End process;`
	`Process (val1, val2, val3, val4, sel)`
	`Begin`
	`with sel select q <= val1 when "00",`
	`val2 when "01",`
	`val3 when "10",`
	`val4 when "11";`
	`end process;`

Simplified Case Expression Support

After associating values '1' and '0' to the first and second elements in an array for a case expression, other elements can be assigned to value '1' provided the case expression is locally static. Prior to VHDL 2008, assigning values to elements after the first and second assignments was not possible. Thus, the following examples are now valid with VHDL 2008.

Example

VHDL 2002	VHDL 2008
Variable s: bit_vector (3 downto 0); Variable c: bit; Subtype bv5 is bit_vector(4 downto 0); …. Case (bv5'(s &c)) When "00000" => … When "00001" => …. When others => …. End case	Variable s: bit_vector (3 downto 0); Variable c: bit; …. Case (s &c) When "00000" => … When "00001" => …. When others => …. End case
Variable s: bit_vector(3 downto 0); …. Case (s) When "0001" => … When "0010" => …. When others => …. End case	Variable s: bit_vector(3 downto 0); …. Case (s) When (0 => '1' others => '0') => … When (1 => '1' others => '0') => …. When others => …. End case

Unconstrained Element Support

VHDL 2008 allows array elements to be unconstrained. It is not necessary to define an array's size and this can instead be determined during signal/variable/constant declaration during elaboration. The keyword open can be used to skip constraints during declaration.

Example

```
Type M_unconstrained is array (natural range<>, natural range <>) of bit;
Type A_unconstrained is array (character range<>) of M_unconstrained;
Type A1_partially_constrained is array (character range 'a' to 'z') of
M_unconstrained;
Subtype s4 is A_unconstrained(open)(0 to 7, 31 downto 16);
```

Note

Support for unconstrained elements currently exists only for arrays and not for records.

Context Declarations

In VHDL 2008, you can create context declarations of your libraries and 'use' clauses.

Example

```
context widget_context is
library IEEE;
use IEEE.std_logic_1164.all, IEEE.numeric_std.all;
use widget_lib.widget_defs.all;
use widget_lib.widget_comps.all;
end context widget_context;
...........
library widget_lib;
context widget_lib.widget_context;
```

Extensions to Generate

VHDL 2008 introduces support for if-else-if and case generate statements. Previously, the generate feature was restricted to the if statement only.

Example

VHDL 2002	VHDL 2008

`for I in width -1 downto 0 generate begin if I = width - 1 generate adder_1: adder(in1, in2, out1); end generate`
 	
`if not I = width - 1 generate adder_2: adder(in3[i],in2[i], out2); end generate end generate`
	`for I in width -1 downto 0 generate begin if I = width - 1 generate adder_1: adder(in1, in2, out1); else generate adder_2: adder(in3[i],in2[i], out2); end generate end generate`

Standard Packages

VHDL 2008 LRM provides updates to the existing language predefined packages namely the Standard, Std_logic_1164, Numeric_bit and Numeric_std Packages. VHDL 2008 also added two new packages namely the Numeric Unsigned Packages (numeric_std_unsigned and numeric_bit_unsigned) which defines the unsigned type and associated operations for a situation when it is required to model a std_ulogic_vector/bit_vector represented as a binary coded number.

Refer to VHDL 2008 LRM section 16.8 for details regarding the enhancements done to Standard package

Updates in Standard Package

VHDL 2008 supports updates to the standard package.

- New data types
  - boolean_vector
  - integer_vector
  - real_vector
  - time_vector
- New Predefined operators on the new and existing data types
  - Relational ("=", "/=", "<", ">", "<=", ">=")
  - Concatenation (“&”)
  - Matching relational ("?=", "?/=", "?>", "?>=", "?<", "?<=")
  - Conditional operators (“??”)
- New functions on the new data types
  - rising_edge()
  - falling_edge()
  - maximum()
  - minimum()

Tip

Limitation: The following operators are not supported: mod, rem on time type, To_string, To_bstring, to_ostring, and to_hstring.

Updates in Std_logic_1164 Package

Predefined operators

- array scalar operators (and, nand, or, nor, xor, xnor)
- Logical reduction operators (and, nand, or, nor, xor, xnor)
- Shift and Rotate operators (sll, srl, rol, ror)
- Conversion functions: TO_01().
- Conditional operator ("??")
- Alias (TO_BV, TO_SLV, TO_SULV, TO_BIT_VECTOR, TO_STD_LOGIC_VECTOR, TO_STD_ULOGIC_VECTOR)

Tip

Limitation: String conversions and TextIO functions are not supported.

Updates in Numeric packages

VHDL 2008 adds support for numeric packages.

New package Numeric_bit_unsigned

- Provides convenient interpretation mechanism for bit_vector represented as binary coded number
- Provided conversion functions and arithemetic operator/functions

New package Numeric_std_unsigned

- Provides convenient interpretation mechanism for std_ulogic_vector represented as binary coded number
- Provided conversion functions and arithemetic operator/functions

New operators in Numeric_bit and Numeric_std package like

- array scalar operators (and, nand, or, nor , xor, xnor)
- logical reduction operators (and, nand, or, nor , xor, xnor)
- overloaded shift operators (sla, sra)

- New functions
  - find_leftmost()
  - find_rightmost()
  - maximum()
  - minimum()
- Overloading of matching relational operators ("?=", "?/=", "?>", "?>=", "?<", and "?<=")
- Conversion Function (is_X)

Tip

Limitation: TextIO and String conversion functions are not supported.

Fixed Point Package

VHDL 2008 adds support for Fixed Point Package which provides a representation of floating numbers in the form of fixed point representation.

The types supported in fixed point package are **ufixed** (unsigned fixed point) and **sfixed** (signed fixed point) types. These types are always supposed to have “downto” ranges with positive ranges for integral part and negative ranges for fractional part. This is very useful for DSP applications. Annex G of the LRM 1076-2008 contains more information about the usage of this package.

Example

6.5 is represented as

```
signal y: ufixed(4 downto -5);
…....
Y <= “0011010000”; -- 6.5
```

Also, since package generics are not supported currently, support for fixed_generic_pkg does not exist currently.

Float Point Package

VHDL 2008 adds support for Floating Point Math Package which allows to represent non integral values with constant absolute precision over a given range.

For certain application domains, it is better to use floating point representation for a given number of bit representation. Floating point values are as per IEEE Std 743 and IEEE Std 854 with a sign bit, exponential field and a fraction field. For more details please refer to section 16.11 of VHDL IEEE 1076-2008 Language Reference Manual

- New data types (float, float32, float64, float128, valid_fpstate
- Predefined operators on the new types
  - Arithmetic operators & functions
  - Comparison operators & functions
  - Relational operators
  - Logical operators
  - Logical reduction operators
  - Matching relational operators
  - Matching functions
  - Maximum and Minimum functions
  - Find_leftmost and Find_rightmost functions
  - Conversion functions
  - Break_number, normalize functions
  - Functions to return constants

Tip

Limitation: String and TextIO functions are not supported.

Example

```
--Signal declaration of a predefined type (float32)
Signal myfloat : float32;
-- Subtype declaration using unconstrained type (float)
Subtype my_float1128 is float(15 down to -112);
-- Operator usage on predefined type (float32)
variable a: float32;
variable b: float32;
signal c, d : float32;
c <= A+B;
d <= 6.5;
```

Expressions Port Map

VHDL 2008 allows input ports to be connected to expressions instead of the need to create temporary signals.

Example

```
Inst1 : trial_ent port map (cin1 => CONV_INTEGER(STD_LOGIC_VECTOR(tdin1)
or x"1010") , cin2 => tdin2, cout1 => tdout1);
```

Read Out Ports

VHDL 2008 allows output ports to be read. Though this feature is mainly intended to help the verification of assertions and monitors, language semantics do not stop it from being used in algorithmic behavior.

The scope of this feature will support reading output ports in architectures, sub-programs etc. In case of variables in sub-programs, the value propagation won't be done as per LRM but in case of signals the connection will be as passed by reference.

Table D-1. Read Out Ports

VHDL 2002	VHDL 2008

`Entity dff is`
`  Port (clk, d: in bit; q, q_n: out bit);`
`End entity;`
 	
`Architecture rtl of dff is`
`Signal q_int:  bit;`
`Begin`
`  Process (clk)`
`  Begin`
`  If (clk = '1') then`
`  q_int <= d;`
`  end if;`
`  end process;`
`  q <= q_int;`
`  q_n <= q_int;`
`end rtl;`
	`Entity dff is`
	`  Port (clk, d: in bit; q, q_n: out bit);`
	`End entity;`
	
	`Architecture rtl of dff is`
	`Begin`
	`  Process (clk)`
	`  Begin`
	`  If (clk = '1') then`
	`  q <= d;`
	`  end if;`
	`  end process;`
	`  q_n <= q;`
	`end rtl;`

Simplified Sensitivity List

VHDL 2008 introduces the keyword 'all' which can be used in process sensitivity list. This will ensure that all signals are appropriately added in the sensitivity list.

Example

Table D-2. Simplified Sensitivity List

VHDL 2002	VHDL 2008

`Process(in1, in2, in3, in4)`
`Begin`
`  Out1 <= in1;`
`  Out2 <= in2;`
`  Out3 <= in3;`
`  Out4 <= in4;`
`End process`
	`Process (all)`
	`Begin`
	`  Out1 <= in1;`
	`  Out2 <= in2;`
	`  Out3 <= in3;`
	`  Out4 <= in4;`
	`End process`

Block Comments

VHDL 2008 introduces C-style multi-line comments starting with '/*' and ending with '*/'. The nesting of block comments is, however, not allowed in VHDL 2008.

Example

Table D-3. Block Comments

VHDL 2002	VHDL 2008

Entity dff is
  Port (clk, d: in bit; q, q_n:
out bit);
End entity;
 	
Architecture rtl of dff is
Signal q_int: bit;
Begin
  Process (clk)
-- This line
-- and this line
-- must be commented
  Begin
  If (clk = '1') then
  q_int <= d;
  end if;
  end process;
  q <= q_int;
  q_n <= q_int;
end rtl;
	Entity dff is
	  Port (clk, d: in bit; q, q_n:
	out bit);
	End entity;
	
	Architecture rtl of dff is
	Signal q_int: bit;
	Begin
	  Process (clk)
	/* This line
	 and this line
	 must be commented*/
	  Begin
	  If (clk = '1') then
	  q_int <= d;
	  end if;
	  end process;
	  q <= q_int;
	  q_n <= q_int;
	end rtl;

Matching Case Statement

In VHDL 2008, case statements are introduced with Don't Care in the choice statement by usage of 'case?'. A simple example is described here where the value of the case expression Xin is matched with expressions containing ‘don't care’. The restriction on Xin is that its value cannot contain a '-' which is an error.

It is also an error if any two choice lists contain the same value. Therefore, the values "---1" and "--1-" in choice lists should yield an error due to an overlap of choices. The case expression is bit/std_ulogic or its vector.

Example

```
case? Xin is
        when "--1" =>
            Zout <= Ain;
        when "-10" =>
            Zout <= Bin;
when others =>
          Zout <= Cin;
```

Array-Scalar Operators

Array Scalar Operators allow a scalar operand to be applied to all the elements of an array. The array-scalar logical operators are AND, OR, NAND, NOR, XOR and XNOR for bit, Boolean and std_ulogic types.

In the example shown here, the AND operator is applied to each bit of A with ASel and result is stored in T.

```
A  : IN BIT_VECTOR (3 downto 0);
ASel  : In BIT;
T <= A and ASel;
```

Logical Reduction Operators

Logical Reduction Operators allow an array to be reduced to array element type by the application of the operator. The logical reduction operators AND, OR, NAND, NOR, XOR and XNOR are pre-defined for bit and Boolean vectors. They must be defined in std_logic_1164 package for std_ulogic_vector and std_logic_vector.

The following is an example of parity bit calculation in both VHDL 2008 and VHDL 2002.

Table D-4. Logical Reduction Operators

VHDL 2002	VHDL 2008
`Parity <= data(0) xor data(1) xor data(2) xor data(3);`	`Parity <= xor data;`

Matching Relational Operators

VHDL 2008 introduces relational operators that return the result of the bit type or std_ulogic type instead of Boolean result. The matching relational operators are "?=", "?/=", "?<", "?<=", "?>" and "?>=". The following example demonstrates the use of matching relational operators which don't require the use of a when-else statement as was the case prior to VHDL-2008.

Table D-5. Matching Relational Operators

VHDL 2002	VHDL 2008
`Control_sig <= '1' when x = y else '0'`	`Control_Sig <= x ?= y;`

Conditional Operator Support

VHDL 2008 adds a conditional operator "??" which converts bit/std_ulogic type to boolean type. This simplifies the task of writing conditional expressions. The operator is implicitly inferenced in expressions by applying the operator tothe entire expression. The "??" operator can be used in ‘until’, ‘if’, ‘elseif’, ‘while’ and ‘when’ statements.

The following example shows implicit inference of the conditional operator.

Table D-6. Conditional Operator Support

VHDL 2002	VHDL 2008

`if a ='1' and b = '1' then`
`  Q  <= d;`
`end if;`
	`if a and b then`
	`  Q <= d;`
	`end if;`

Maximum and Minimum Function Support

VHDL 2008 introduces pre-defined maximum and minimum functions with semantics "[ScalarType, ScalarType] return ScalarType" and "[DiscreteArrayType, DiscreteArrayType] return DiscreteArrayType".

Maximum and minimum functions make use of the "<" operator for ordering. VHDL 2008 also predefines maximum and minimum functions as reduction operators on array values with semantic "[ArrayType] return ArrayElementType".

Unconstrained Record Elements

VHDL 2008 allows you to keep record elements unconstrained during type definition. These can be constrained later during elaboration or signal declaration.

Example

```
type myrec is record
a : std_logic_vector;
b : bit;
c : bit_vector;
end record;

signal s : myrec((3 downto 0), open, (2 downto 0));
```

In this example, the constraints of **‘a’** and **‘c’** elements are defined only during signal declaration and not during type definition.

Type Generics

VHDL 2008 enables packages/entity/subprograms and so on which can have generics with values and types. Package generics allows you to instantiate the package with an override value or type to allow the reuse of a package in different contexts. With the support of this feature Precision now handles both values and types specification for package generics.

Example

Package Declaration

```
library IEEE;
use IEEE.std_logic_1164.all;

package my_pack is
  type mytype is array (7 downto 0) of std_logic;
end package;

library IEEE;
use IEEE.std_logic_1164.all;
package pack is
  generic(width : integer := 3;
          in0 : std_logic_vector (3 downto 0) := "1011";
          vector : integer := 7;
          type T);
    procedure proc(a : in T; b : out T);
end package;

package body pack is
  procedure proc(a : in T; b : out T) is
  begin
      b := a;
  end proc;

end package body;
```

Package Instantiation

```
library IEEE;
use IEEE.std_logic_1164.all;
package pack_inst is new work.pack generic map
                                        (7,"1011",3,work.my_pack.mytype);
```

Tip

Limitation: Package generics are supported within the scope of global declarations. Package declaration and instantiation within another package is not supported. Usage of generic types in contexts of entity and subprogram is not supported.

Generic List

VHDL 2008 enables extension to generic list (declared inside allowed declaration scope namely the subprogram, component, design entity, block declaration or package) so to allow a name declared as part of generic be used in other declarations.

Example

```
library ieee;
use ieee.std_logic_1164.all;

package I2C_IOE_PCK is
type t_Device_Data_Array is array (natural range <>) of
std_logic_vector(15 downto 0);

component I2C_IOE is
generic
(
  c_NrOffDevices : natural := 3;
  c_Port_RW_Config : t_Device_Data_Array(c_NrOffDevices -1 downto 0)
);
Port
(
  DataIn : in t_Device_Data_Array (c_NrOffDevices -1 downto 0);
  DataOut : out t_Device_Data_Array (c_NrOffDevices -1 downto 0)
);
end component;
end package I2C_IOE_PCK;


use work.I2C_IOE_PCK.all;

entity I2C_IOE is
generic
(
  c_NrOffDevices : natural := 3;
  c_Port_RW_Config : t_Device_Data_Array(c_NrOffDevices -1 downto 0)

);
Port
(
  DataIn : in t_Device_Data_Array (c_NrOffDevices -1 downto 0);
  DataOut : out t_Device_Data_Array (c_NrOffDevices -1 downto 0)
);
end I2C_IOE;

architecture Behavioral of I2C_IOE is
begin
  DataOut <= DataIn;
end Behavioral;
```

In this example, the generic c_NrOffDevices is being used in the context of declaring other generics, port declaration, and so on.

Bit String Literal

VHDL 2008 enhances bit-string literal support by allowing the following:

1. Size Specification which is an integer value defined as length of the bit string and it is optional.

2. Usage of any graphic character as part of the bit value.
3. Base specification which can be of following forms and have their predefined expansion and truncation rules:

   B - Binary

   O - Octal

   X - Hexadecimal

   UB - Unsigned Binary

   UO - Unsigned Octal

   UX - Unsigned Hexadecimal

   SB - Signed Binary

   SO - Signed Octal

   SX - Signed Hexadecimal

   D - Decimal

Example

```
package CONV_PACK_constant7 is
 type row is array ( 0 to 3 ) of bit;
 type matrix is array ( 0 to 3 ) of row;
end CONV_PACK_constant7;

use work.CONV_PACK_constant7.all;
entity constant7 is
port( in0, in1 , in2 , in3 : row;
      control              : integer range 0 to 3;
      out_mux              : out row);
end;

architecture constant7 of constant7 is
 signal in_matrix1 : matrix;
 constant in_matrix : matrix :=
(4SX"f9",4SB"00000011",4UB"0000001010",4UO"06");
begin
 process(in0, in1, in2, in3, control,in_matrix1)
 begin
  in_matrix1 <=  (0 => in0 , 1 => in1 , 2 => in2 , 3 => in3);
  out_mux   <=  in_matrix( control ) xor in_matrix1(control);
 end process;
end;
```

Example

```
library IEEE;
use IEEE.std_logic_1164.all;
package CONV_PACK_component is
attribute ENUM_ENCODING : STRING;
type myenum is (S1, S3, C4, C7, C3, A1, D9, K9, R2);
attribute ENUM_ENCODING of myenum : type is
   "0000 0001 0010 0011 0100 0101 0110 0111 1000";
   function std_logic_vector_to_myenum(arg : in std_logic_vector( 1 to 4
))
               return myenum;
end CONV_PACK_component;

package body CONV_PACK_component is

   function std_logic_vector_to_myenum(arg : in std_logic_vector( 1 to 4
))
   return myenum is
   begin
      case arg is
         when 4D"0" => return S1;
         when 4D"1" => return S3;
         when 4D"2" => return C4;
         when 4D"3" => return C7;
         when 4D"4" => return C3;
         when 4D"5" => return A1;
         when 4D"6" => return D9;
         when 4D"7" => return K9;
         when 4D"8" => return R2;
         when others => return S1;
      end case;
   end;
end CONV_PACK_component;
```

Resolved Element Support

In VHDL 2002, resolved signals and subtypes are used to specify multiply driven sources.

VHDL 2008 allows you to specify the resolution function for composite types (records or arrays). In VHDL 2008, the predefined type std_logic_vector is now made a subtype of std_ulogic_vector is defined as:

```
subtype std_logic_vector is (resolved) std_ulogic_vector
```

This definition implies the following:

- It is possible to assign a std_ulogic_vector to std_logic_vector and vice versa.
- It is possible to specify the resolution function for each element of the std_logic_vector, and the multiple drivers to signals of std_logic_vector element types would get resolved with resolution function "resolved".

Resolution function notes to be considered include:

- Resolution functions are relevant when multiple sources drive a signal that has a resolution marking.
- Resolution function indications can be given at a signal or subtype declaration. In case of a resolution function indication on a signal as well as subtype, the one at the signal will have higher priority than the subtype. The resolution indication closest to the top in the type hierarchy of the signal overrides all other resolution indications.
- Resolution functions are implicitly called.
- Resolution functions can be of the following types:
  - pragma based (wired or, wired and, three state)
  - user defined

Precision support of resolution functions consists of the following:

- All predefined array types (std_logic_vector, ufixed, sfixed, UNSIGNED, SIGNED, float) where resolution function indication can be specified for its elements. These predefined types are part of the IEEE 2008 packages.
- Resolution function indication on user defined subtypes, signals of predefined types

```
subtype arr_slv is (my_res_func1) array (natural range <>) of
std_logic_vector;
signal s: arr_slv1; --elements will get resolved with my_res_func1
function
signal t : (my_res_func2) std_logic_vector; --elements will get
resolved with my_res_func2 function
```

- Resolution function indication specification at multiple levels for user defined types, subtypes and signals. The number of parenthesis around the resolution function defines the nesting level to which resolution function is to be applied.

```
type arr_slv is array (natural range <>) of std_logic_vector;
subtype res_arr_slv is ((my_res_func3)) arr_slv;  --element of type
std_logic will get resolved with my_res_func3 function
signal u : ((my_res_func4)) arr_slv (3 downto 0); --element of type
std_logic will get resolved with my_res_func4 function
```

- Resolution function indication specification for record types, subtypes and signals at multiple levels.

```
TYPE E IS ( '0', '1', '2', '3', '4', '5',  '6', '7' );
type E_ARR is array (natural range <>) of E;
subtype E_ARR_RES is (f1) E_ARR; --element of type 'E' will get
resolved with function f1

type E_ARR_ARR is array (natural range <>) of E_ARR;
subtype E_ARR_ARR_RES is ((f2)) E_ARR_ARR; --element of type 'E'
will get resolved with function f2

type S is record
  w : E_ARR_ARR(1 downto 1)(2 downto 2);
  x : std_ulogic;
end record;

subtype S_RES is ( w ((f3)),  x f4 ) S; --element of type 'E' in
record field w will get resolved with function f3 and element of
type 'std_ulogic' in record field x will get resolved with function
f4.
```

Example

```
library ieee;
use ieee.std_logic_1164.all;

package mypack is

    type mytype is ('0','1');
    type new_myType1D  is array (integer range <>) of mytype;

    function res_1D(s:new_myType1D)return mytype;
    function res1_1D(s:new_myType1D)return mytype;

    type my2D is array (NATURAL range <>) of new_myType1D;
    type myMD is array (NATURAL range <>,NATURAL range <>) of mytype;
    subtype my1darr is (res_1D) new_myType1D;
    subtype my2darr is ((res_1D)) my2D;
    subtype myMultarr is (res_1D) myMD;

    function my_and(s,r:mytype)return mytype;
    function my_or(s,r:mytype)return mytype;
    function res(s:new_myType1D)return mytype;
    function res1(s:new_myType1D)return mytype;

    subtype mytype1 is (res) new_myType1D;
    subtype mytypeR is res mytype;
    subtype myType1D is (res) new_myType1D;

    subtype sub is bit_vector(3 downto 0);
    type arr is array (natural range <>) of sub;
    function res_func(in1: arr) return sub;
    subtype subr is res_func sub;


    function conv_slv21D(slv: std_logic_vector) return new_myType1D;
    function conv_1D2slv(type1D:new_myType1D) return std_logic_vector;
    function conv_2Dtoslv(type2D:my2darr) return std_logic_vector;
    function conv_slvto2D(slv:std_logic_vector) return my2darr;
    function conv_slvtoM2D(slv:std_logic_vector) return myMultarr;
    function conv_M2Dtoslv(type2D:myMultarr) return std_logic_vector;
end;

package body  mypack is

    function conv_slv21D(slv: std_logic_vector) return new_myType1D is
        variable result:new_myType1D(5 downto 0);
    begin
        for i in slv'range loop
            if(slv(i) = '0') then
                result(i) := '0';
            end if;
            if (slv(i) = '1') then
                result(i) := '1';
            end if;
        end loop;
        return result;
    end conv_slv21D;
```

```
function conv_1D2slv(type1D:new_myType1D) return std_logic_vector is
    variable result:std_logic_vector(5 downto 0);
begin
    for i in type1D'range loop
        if(type1D(i) = '0') then
            result(i) := '0';
        end if;
        if (type1D(i) = '1') then
            result(i) := '1';
        end if;
    end loop;
    return result;
end conv_1D2slv;

function conv_2Dtoslv(type2D:my2darr) return std_logic_vector is
    variable result:std_logic_vector(24 downto 0);
begin
    for i in type2D'range loop
        for j in type2D(i)'range loop
            if(type2D(i)(j) = '0') then
                result(i*5+j) := '0';
            end if;
            if (type2D(i)(j) = '1') then
                result(i*5 +j) := '1';
            end if;
        end loop;
    end loop;
    return result;
end conv_2Dtoslv;

function conv_slvto2D(slv:std_logic_vector) return my2darr is
    variable result:my2darr(4 downto 0)(4 downto 0);
    variable temp:integer := 0;
begin
    for i in slv'range loop
        if(slv(i) = '1') then
        result(i/5)(i mod 5) := '1';
        end if;
        if(slv(i) = '0') then
        result(i/5)(i mod 5) := '0';
        end if;

    end loop;
    return result;
end conv_slvto2D;

function conv_M2Dtoslv(type2D:myMultarr) return std_logic_vector is
    variable result:std_logic_vector(24 downto 0);
begin
    for i in 0 to 4 loop
        for j in 0 to 4  loop
            if(type2D(i ,j) = '0') then
                result(i*5+j) := '0';
            end if;
            if (type2D(i ,j) = '1') then
                result(i*5 +j) := '1';
            end if;
```

```
        end loop;
    end loop;
    return result;
end conv_M2Dtoslv;


function conv_slvtoM2D(slv:std_logic_vector) return myMultarr is
    variable result:myMultarr(4 downto 0,4 downto 0);
    variable temp:integer := 0;
begin
    for i in slv'range loop
        if(slv(i) = '1') then
            result(i/5 , i mod 5) := '1';
        end if;
        if(slv(i) = '0') then
            result(i/5 , i mod 5) := '0';
        end if;
    end loop;
    return result;
end conv_slvtoM2D;



function my_and(s,r:mytype)return mytype is
    variable result : mytype := '0';
begin
    if (s = '1' and r = '1') then
        result := '1';
    end if;
    return result;
end my_and;

function my_or(s,r:mytype)return mytype is
    variable result : mytype := '1';
begin
    if (s = '0' and r = '0') then
        result := '0';
    end if;
    return result;
end my_or;

function res(s:new_myType1D)return myType is
    variable result:myType:='1';
begin
    for i in s'range loop
        result:= my_and(result,s(i));
    end loop;
    return result;
end res;

function res1(s:new_myType1D)return mytype is
    variable result:mytype:='0';
begin
    for i in s'range loop
        result:= my_or(result,s(i));
    end loop;
    return result;
```

```
    end res1;

    function res_1D(s:new_myType1D)return mytype is
        variable result:mytype:='0';
    begin
        for i in s'range loop
            result:= my_or(result,s(i));
        end loop;
        return result;
    end;

    function res1_1D(s:new_myType1D)return mytype is
        variable result:mytype:='0';
    begin
        for i in s'range loop
            result:= my_or(result,s(i));
        end loop;
        return result;
    end;
    function res_func(in1: arr) return sub is
        variable var : sub := in1(in1'low);
    begin
        for i in in1'low  + 1 to in1'high loop
            var(3) := var(3) or in1(i)(3);
            var(2) := var(2) or in1(i)(2);
            var(1) := var(1) or in1(i)(1);
            var(0) := var(0) or in1(i)(0);
        end loop;
        return var;
    end;

end;

use work.mypack.all;
library ieee;
use ieee.std_logic_1164.all;
entity test is
    generic(
             P : NATURAL  range 0 to 1  := 1
            );
    port (in1, in2 : in std_logic_vector(24 downto 0);
          out1: out std_logic_vector(24 downto 0)
          );
end;

architecture arch of test is
    signal a: (res)new_myType1D(5 downto 0);
    signal b: (res)new_myType1D(5 downto 0);
    signal o: (res)new_myType1D(5 downto 0);

    signal a2: (res_1D)myMultarr(4 downto 0,4 downto 0);
    signal b2: (res_1D)myMultarr(4 downto 0,4 downto 0);
    signal c2: (res_1D)myMultarr(4 downto 0,4 downto 0);

begin
    a2 <= conv_slvtoM2D(in1);
    b2 <= conv_slvtoM2D(in2);
```

```
        c2 <= a2;
        G: if(P = 5) generate
            c2 <= b2;
        end generate G;

        out1 <= conv_M2Dtoslv(c2);

    end;
```

Appendix E
readVSDC Flow File

A readVSDC flow file enables in-line reading of guide files from both the Oasys-RTL and Design Compiler tools. Combined with the new Heuristic Name Lookup function that performs name translation at runtime, test performance with Oasys-RTL and Design Compiler guide files is greatly improved.

Using the readVSDC Flow

The readVSDC flow enables in-line reading of guide files from both the Oasys-RTL and Design Compiler tools.

The “-flow readVSDC” option extracts duplicate and constant assertion constraints from the Synopsys VSDC file (or Mentor Oasys). Duplicate information is quite critical for LEC, and constant assertions are also discovered automatically. However, in many deep cases, the automatic process requires additional help, and the VSDC file should be providing a complete record. Both duplicate and constant settings are verified in FormalPro with a proof, and an error will be reported if they are not valid.

Procedure

1. Set the $FORMALPRO_VSDCFILE environment variable to the guide file name before invoking FormalPro. For example:

   ```
   setenv FORMALPRO_VSDCFILE  path/myfile.vsdc
   export FORMALPRO_VSDCFILE=path/rt_opt_reg.txt
   ```

2. Invoke FormalPro with multiple flow files as needed. For example:

   ```
   formalpro -flow dc_ultra -flow readVSDC -a rtl.fl -b netlist.fl
   formalpro -flow oasys -flow readVSDC -a rtl.fl -b netlist.fl
   ```

   Depending on the flow, a local constraint file is generated that you can edit for direct input to FormalPro:

   - **Design Compiler flow —** vsdc.constraint is created
   - **Mentor Graphics Oasys-RTL flow —** _rtAuTo.constraint is created

   The time stamp prevents overwriting edited files.

Caution

The local constraint translations files are overwritten if the $FORMALPRO_VSDCFILE environment variable points to a file that is newer than the local file, so manage these files with care.

Examples

This is how the command line with the "-flow readVSDC" executes the following command internally:

```
formalpro -flow dc_ultra -flow readVSDC ...
```

The internal "readVSDC" executes this command:

```
$MGC_HOME/bin/fp_utility vsdc2constraint -quiet -stdout  -fast
```

(where $FORMALPRO_VSDCFILE points to the input file and path.)

The preceding command uses the "-fast" mode, which is just a default translation of strings. It does not perform a lookup of actual names. In this mode, the resulting translation is written to a file in the local directory. The local file can then be edited. One solution is to modify that file and use it directly as a parameter to the -constraintFile option. Copy and rename this file (*vsdc.constraint*) before running the following manual command to compare them.

If you run the following manual command with a *formalpro.cache* that has already been run through the match phase, then a lookup function will happen, which attempts to convert the Synopsys name to the actual "best fit" FormalPro name. It does not set the -fast option.

```
$MGC_HOME/bin/fp_utility vsdc2constraint $FORMALPRO_VSDCFILE \
   -cache formalpro.cache
```

This makes an output file *vsdc.constraint* that can be examined. A best practice is to simply extract the lines that are improved in this output file and add them as an extra constraint. Otherwise, if you prefer to use the entire file, do not use the "-flow readVDSC" command; use "-constraintFile vsdc.constraint" instead.

A guide_change_names block is also included in the Synopsys VSDC file. It is often useless and trivial and it would add runtime to FormalPro to implement the commands, but in the following example, the flow requires unconventional names:

```
$MGC_HOME/bin/fp_utility vsdc2constraint $FORMALPRO_VSDCFILE \
   -cache formalpro.cache -change_names
```

In some flows, the change_names command is used more than once and the resulting VSDC file can have two transitions. The following is a complex example:

```
guide_change_names \
    { cell s_data_ reg[1][0] s_data _regx1xx0x } \
```

followed by:

```
{ cell s_data _regx1xx0x s_data_reg_255 } \
```

Because of the XX bus naming and it appears this is the basis of the last transition, the RTL compiler needs to name the arrays with style regx2x.

Add the following to the **formalpro** command:

```
-a \
-latchInstMemoryFormat %s_latx%dxx%dx \
-dffInstMemoryFormat %s_regx%dxx%dx \
-latchInstVectorFormat %s_latx%dx \
-dffInstVectorFormat %s_regx%dx \
-arrayNameFormat %sx%dx\
```

The following sample run required approximately 15 minutes, with 182K lines in the file:

```
%  cp -r /path-DUT/formalpro.cache .
%  fp_utility VSDC data.vsdc -change_names -noreg_const -nodup
#Input file is    data.vsdc
#    Found 0 guide_inv_push terms in data.vsdc file
#    Found 2 guide_change_names terms in data.vsdc file
#    Found 0 guide_reg_merging terms in data.vsdc file
#    Reading matched_objects.... ./formalpro.cache/reports/matched_objects.report
#        Processing VSDC lines ...
#
#Found 0 of 0 reg_constant assignments in VSDC file, ignored 0, 0 were X(dont-care)
#Found 0 of 0 duplicate assignments in VSDC file
#Constraint output file is   vsdc.constraint
#Found 66374 non-trivial change_names items from 66374 gross
#Change_names output file is   vsdc_change_names.rule
#Copied ./formalpro.cache/inputFiles/FormalPro.rule for insertion of
change_name_rules
#File vsdc_change_names.rule can be used as a full match rule file. use:
formalpro -rulefile vsdc_change_names.rule
```

Note the last line of comments. That comment is correct, but the file must be edited for at least one more rule. Adding a pre-process rule that makes the FormalPro database name fit the expectation of the VSDC initial transform.

```
# … bottom of the new rule file ….
########################################
###    BEST PLACE FOR USER ADD-ONS    ###
########################################
# first correct the Generate Loop Index data_0_reg(i)(j) -> data_reg(i)(j)
Ar s/data\.[0-9]*\./data./
#formalpro match rules derived from VSDC guide_change_names
#Place after ### Best Place marker
B s/s\.data.9162/s.data.2xx464x/g
…….
# Remainder of the transforms and end of file.
```

Index

— Q —

— R —

— S —

End-User License Agreement

The latest version of the End-User License Agreement is available on-line at:
www.mentor.com/eula

IMPORTANT INFORMATION

USE OF ALL SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE PRODUCTS. USE OF SOFTWARE INDICATES CUSTOMER'S COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.

END-USER LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Software (as defined in Section 2) and hardware (collectively "Products") between the company acquiring the Products ("Customer"), and the Mentor Graphics entity that issued the corresponding quotation or, if no quotation was issued, the applicable local Mentor Graphics entity ("Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by Customer and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If Customer does not agree to these terms and conditions, promptly return or, in the case of Software received electronically, certify destruction of Software and all accompanying items within five days after receipt of Software and receive a full refund of any license fee paid.

1. ORDERS, FEES AND PAYMENT.

1.1. To the extent Customer (or if agreed by Mentor Graphics, Customer's appointed third party buying agent) places and Mentor Graphics accepts purchase orders pursuant to this Agreement (each an "Order"), each Order will constitute a contract between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of this Agreement, any applicable addenda and the applicable quotation, whether or not those documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order or presented in any electronic portal or automated order management system, whether or not required to be electronically accepted, will not be effective unless agreed in writing and physically signed by an authorized representative of Customer and Mentor Graphics.

1.2. Amounts invoiced will be paid, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Prices do not include freight, insurance, customs duties, taxes or other similar charges, which Mentor Graphics will state separately in the applicable invoice. Unless timely provided with a valid certificate of exemption or other evidence that items are not taxable, Mentor Graphics will invoice Customer for all applicable taxes including, but not limited to, VAT, GST, sales tax, consumption tax and service tax. Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such taxes imposed on payments by Customer hereunder will be Customer's sole responsibility. If Customer appoints a third party to place purchase orders and/or make payments on Customer's behalf, Customer shall be liable for payment under Orders placed by such third party in the event of default.

1.3. All Products are delivered FCA factory (Incoterms 2010), freight prepaid and invoiced to Customer, except Software delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor Graphics retains a security interest in all Products delivered under this Agreement, to secure payment of the purchase price of such Products, and Customer agrees to sign any documents that Mentor Graphics determines to be necessary or convenient for use in filing or perfecting such security interest. Mentor Graphics' delivery of Software by electronic means is subject to Customer's provision of both a primary and an alternate e-mail address.

2. **GRANT OF LICENSE.** The software installed, downloaded, or otherwise acquired by Customer under this Agreement, including any updates, modifications, revisions, copies, documentation, setup files and design data ("Software") are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors, who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Except for Software that is embeddable ("Embedded Software"), which is licensed pursuant to separate embedded software terms or an embedded software supplement, Mentor Graphics grants to Customer, subject to payment of applicable license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form (except as provided in Subsection 4.2); (b) for Customer's internal business purposes; (c) for the term of the license; and (d) on the computer hardware and at the site authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Customer may have Software temporarily used by an employee for telecommuting purposes from locations other than a Customer office, such as the employee's residence, an airport or hotel, provided that such employee's primary place of employment is the site where the Software is authorized for use. Mentor Graphics' standard policies and programs, which vary depending on Software, license fees paid or services purchased, apply to the following: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be technically implemented through the use of authorization codes or similar devices); and (c) support services provided, including eligibility to receive telephone support, updates, modifications, and revisions. For the avoidance of doubt, if Customer provides any feedback or requests any change or enhancement to Products, whether in the course of receiving support or consulting services, evaluating Products, performing beta testing or otherwise, any inventions, product improvements, modifications or developments made by Mentor Graphics (at Mentor Graphics' sole discretion) will be the exclusive property of Mentor Graphics.

3. **BETA CODE.**

3.1. Portions or all of certain Software may contain code for experimental testing and evaluation (which may be either alpha or beta, collectively "Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to Customer a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. Mentor Graphics may choose, at its sole discretion, not to release Beta Code commercially in any form.

3.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer's use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer's evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.

3.3. Customer agrees to maintain Beta Code in confidence and shall restrict access to the Beta Code, including the methods and concepts utilized therein, solely to those employees and Customer location(s) authorized by Mentor Graphics to perform beta testing. Customer agrees that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on Customer's feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this Subsection 3.3 shall survive termination of this Agreement.

4. **RESTRICTIONS ON USE.**

4.1. Customer may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. Except for Embedded Software that has been embedded in executable code form in Customer's product(s), Customer shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. Customer shall not make Products available in any form to any person other than Customer's employees and on-site contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Products and ensure that any person permitted access does not disclose or use Products except as permitted by this Agreement. Customer shall give Mentor Graphics written notice of any unauthorized disclosure or use of the Products as soon as Customer becomes aware of such unauthorized disclosure or use. Customer acknowledges that Software provided hereunder may contain source code which is proprietary and its confidentiality is of the highest importance and value to Mentor Graphics. Customer acknowledges that Mentor Graphics may be seriously harmed if such source code is disclosed in violation of this Agreement. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, disassemble, reverse-compile, or reverse-engineer any Product, or in any way derive any source code from Software that is not provided to Customer in source code form. Log files, data files, rule files and script files generated by or for the Software (collectively "Files"), including without limitation files containing Standard Verification Rule Format ("SVRF") and Tcl Verification Format ("TVF") which are Mentor Graphics' trade secret and proprietary syntaxes for expressing process rules, constitute or include confidential information of Mentor Graphics. Customer may share Files with third parties, excluding Mentor Graphics competitors, provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Customer may use Files containing SVRF or TVF only with Mentor Graphics products. Under no circumstances shall Customer use Products or Files or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Products, or disclose to any third party the results of, or information pertaining to, any benchmark.

4.2. If any Software or portions thereof are provided in source code form, Customer will use the source code only to correct software errors and enhance or modify the Software for the authorized use, or as permitted for Embedded Software under separate embedded software terms or an embedded software supplement. Customer shall not disclose or permit disclosure of source code, in whole or in part, including any of its methods or concepts, to anyone except Customer's employees or on-site contractors, excluding Mentor Graphics competitors, with a need to know. Customer shall not copy or compile source code in any manner except to support this authorized use.

4.3. Customer agrees that it will not subject any Product to any open source software ("OSS") license that conflicts with this Agreement or that does not otherwise apply to such Product.

4.4. Customer may not assign this Agreement or the rights and duties under it, or relocate, sublicense, or otherwise transfer the Products, whether by operation of law or otherwise ("Attempted Transfer"), without Mentor Graphics' prior written consent and payment of Mentor Graphics' then-current applicable relocation and/or transfer fees. Any Attempted Transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and/or the licenses granted under this Agreement. The terms of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer's permitted successors in interest and assigns.

4.5. The provisions of this Section 4 shall survive the termination of this Agreement.

5. **SUPPORT SERVICES.** To the extent Customer purchases support services, Mentor Graphics will provide Customer with updates and technical support for the Products, at the Customer site(s) for which support is purchased, in accordance with Mentor Graphics' then current End-User Support Terms located at http://supportnet.mentor.com/supportterms.

6. **OPEN SOURCE SOFTWARE.** Products may contain OSS or code distributed under a proprietary third party license agreement, to which additional rights or obligations ("Third Party Terms") may apply. Please see the applicable Product documentation (including license files, header files, read-me files or source code) for details. In the event of conflict between the terms of this Agreement

(including any addenda) and the Third Party Terms, the Third Party Terms will control solely with respect to the OSS or third party code. The provisions of this Section 6 shall survive the termination of this Agreement.

7. **LIMITED WARRANTY.**

7.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Products, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Products will meet Customer's requirements or that operation of Products will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. Customer must notify Mentor Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Software under an Order and does not renew or reset, for example, with the delivery of (a) Software updates or (b) authorization codes or alternate Software under a transaction involving Software re-mix. This warranty shall not be valid if Products have been subject to misuse, unauthorized modification, improper installation or Customer is not in compliance with this Agreement. MENTOR GRAPHICS' ENTIRE LIABILITY AND CUSTOMER'S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF THE PRODUCTS TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF THE PRODUCTS THAT DO NOT MEET THIS LIMITED WARRANTY. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) PRODUCTS PROVIDED AT NO CHARGE; OR (C) BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."

7.2. THE WARRANTIES SET FORTH IN THIS SECTION 7 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO PRODUCTS PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.

8. **LIMITATION OF LIABILITY.** TO THE EXTENT PERMITTED UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT RECEIVED FROM CUSTOMER FOR THE HARDWARE, SOFTWARE LICENSE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 8 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

9. **THIRD PARTY CLAIMS.**

9.1. Customer acknowledges that Mentor Graphics has no control over the testing of Customer's products, or the specific applications and use of Products. Mentor Graphics and its licensors shall not be liable for any claim or demand made against Customer by any third party, except to the extent such claim is covered under Section 10.

9.2. In the event that a third party makes a claim against Mentor Graphics arising out of the use of Customer's products, Mentor Graphics will give Customer prompt notice of such claim. At Customer's option and expense, Customer may take sole control of the defense and any settlement of such claim. Customer WILL reimburse and hold harmless Mentor Graphics for any LIABILITY, damages, settlement amounts, costs and expenses, including reasonable attorney's fees, incurred by or awarded against Mentor Graphics or its licensors in connection with such claims.

9.3. The provisions of this Section 9 shall survive any expiration or termination of this Agreement.

10. **INFRINGEMENT.**

10.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Product acquired by Customer hereunder infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay costs and damages finally awarded against Customer that are attributable to such action. Customer understands and agrees that as conditions to Mentor Graphics' obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.

10.2. If a claim is made under Subsection 10.1 Mentor Graphics may, at its option and expense: (a) replace or modify the Product so that it becomes noninfringing; (b) procure for Customer the right to continue using the Product; or (c) require the return of the Product and refund to Customer any purchase price or license fee paid, less a reasonable allowance for use.

10.3. Mentor Graphics has no liability to Customer if the action is based upon: (a) the combination of Software or hardware with any product not furnished by Mentor Graphics; (b) the modification of the Product other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of the Product as part of an infringing process; (e) a product that Customer makes, uses, or sells; (f) any Beta Code or Product provided at no charge; (g) any software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; (h) OSS, except to the extent that the infringement is directly caused by Mentor Graphics' modifications to such OSS; or (i) infringement by Customer that is deemed willful. In the case of (i), Customer shall reimburse Mentor Graphics for its reasonable attorney fees and other costs related to the action.

10.4. THIS SECTION 10 IS SUBJECT TO SECTION 8 ABOVE AND STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS, AND CUSTOMER'S SOLE AND EXCLUSIVE REMEDY, FOR DEFENSE,

SETTLEMENT AND DAMAGES, WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY PRODUCT PROVIDED UNDER THIS AGREEMENT.

11. **TERMINATION AND EFFECT OF TERMINATION.**

    11.1. If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized term. Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement immediately upon written notice if Customer: (a) exceeds the scope of the license or otherwise fails to comply with the licensing or confidentiality provisions of this Agreement, or (b) becomes insolvent, files a bankruptcy petition, institutes proceedings for liquidation or winding up or enters into an agreement to assign its assets for the benefit of creditors. For any other material breach of any provision of this Agreement, Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement upon 30 days written notice if Customer fails to cure the breach within the 30 day notice period. Termination of this Agreement or any license granted hereunder will not affect Customer's obligation to pay for Products shipped or licenses granted prior to the termination, which amounts shall be payable immediately upon the date of termination.

    11.2. Upon termination of this Agreement, the rights and obligations of the parties shall cease except as expressly set forth in this Agreement. Upon termination of this Agreement and/or any license granted under this Agreement, Customer shall ensure that all use of the affected Products ceases, and shall return hardware and either return to Mentor Graphics or destroy Software in Customer's possession, including all copies and documentation, and certify in writing to Mentor Graphics within ten business days of the termination date that Customer no longer possesses any of the affected Products or copies of Software in any form.

12. **EXPORT.** The Products provided hereunder are subject to regulation by local laws and European Union ("E.U.") and United States ("U.S.") government agencies, which prohibit export, re-export or diversion of certain products, information about the products, and direct or indirect products thereof, to certain countries and certain persons. Customer agrees that it will not export or re-export Products in any manner without first obtaining all necessary approval from appropriate local, E.U. and U.S. government agencies. If Customer wishes to disclose any information to Mentor Graphics that is subject to any E.U., U.S. or other applicable export restrictions, including without limitation the U.S. International Traffic in Arms Regulations (ITAR) or special controls under the Export Administration Regulations (EAR), Customer will notify Mentor Graphics personnel, in advance of each instance of disclosure, that such information is subject to such export restrictions.

13. **U.S. GOVERNMENT LICENSE RIGHTS.** Software was developed entirely at private expense. The parties agree that all Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to U.S. FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. government or a U.S. government subcontractor is subject solely to the terms and conditions set forth in this Agreement, which shall supersede any conflicting terms or conditions in any government order document, except for provisions which are contrary to applicable mandatory federal laws.

14. **THIRD PARTY BENEFICIARY.** Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, Microsoft Corporation and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.

15. **REVIEW OF LICENSE USAGE.** Customer will monitor the access to and use of Software. With prior written notice and during Customer's normal business hours, Mentor Graphics may engage an internationally recognized accounting firm to review Customer's software monitoring system and records deemed relevant by the internationally recognized accounting firm to confirm Customer's compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FlexNet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics' request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is revealed. Mentor Graphics shall treat as confidential information all information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement. The provisions of this Section 15 shall survive the termination of this Agreement.

16. **CONTROLLING LAW, JURISDICTION AND DISPUTE RESOLUTION.** The owners of certain Mentor Graphics intellectual property licensed under this Agreement are located in Ireland and the U.S. To promote consistency around the world, disputes shall be resolved as follows: excluding conflict of laws rules, this Agreement shall be governed by and construed under the laws of the State of Oregon, U.S., if Customer is located in North or South America, and the laws of Ireland if Customer is located outside of North or South America or Japan, and the laws of Japan if Customer is located in Japan. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of the courts of Portland, Oregon when the laws of Oregon apply, or Dublin, Ireland when the laws of Ireland apply, or the Tokyo District Court when the laws of Japan apply. Notwithstanding the foregoing, all disputes in Asia (excluding Japan) arising out of or in relation to this Agreement shall be resolved by arbitration in Singapore before a single arbitrator to be appointed by the chairman of the Singapore International Arbitration Centre ("SIAC") to be conducted in the English language, in accordance with the Arbitration Rules of the SIAC in effect at the time of the dispute, which rules are deemed to be incorporated by reference in this section. Nothing in this section shall restrict Mentor Graphics' right to bring an action (including for example a motion for injunctive relief) against Customer in the jurisdiction where Customer's place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.

17. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.

18. **MISCELLANEOUS.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements. Any translation of this Agreement is provided to comply with local legal requirements only. In the event of a dispute between the English and any non-English versions, the English version of this Agreement shall govern to the extent not prohibited by local law in the applicable jurisdiction. This Agreement may only be modified in writing, signed by an authorized representative of each party. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.