Formality®
Automated Setup File (SVF)
Manual

Version X-2005.12, December 2005

Comments?
Send comments on the documentation by going to http://solvnet.synopsys.com, then clicking "Enter a Call to the Support Center."

SYNOPSYS®

Copyright Notice and Proprietary Information

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

> "This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____."

Destination Control Statement

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

Disclaimer

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

Registered Trademarks (®)

Synopsys, AMPS, Arcadia, C Level Design, C2HDL, C2V, C2VHDL, Cadabra, Calaveras Algorithm, CATS, CRITIC, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSIM, HSPICE, Hypermodel, iN-Phase, in-Sync, Leda, MAST, Meta, Meta-Software, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PowerMill, PrimeTime, RailMill, RapidScript, Saber, SiVL, SNUG, SolvNet, Superlog, System Compiler, TetraMAX, TimeMill, TMA, VCS, Vera, and Virtual Stepper are registered trademarks of Synopsys, Inc.

Trademarks (™)

Active Parasitics, AFGen, Apollo, Apollo II, Apollo-DPII, Apollo-GA, ApolloGAII, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanTestchip, AvanWaves, BCView, Behavioral Compiler, BOA, BRT, Cedar, ChipPlanner, Circuit Analysis, Columbia, Columbia-CE, Comet 3D, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, Cyclelink, Davinci, DC Expert, DC Expert *Plus*, DC Professional, DC Ultra, DC Ultra Plus, Design Advisor, Design Analyzer, Design Vision, DesignerHDL, DesignTime, DFM-Workbench, Direct RTL, Direct Silicon Access, Discovery, DW8051, DWPCI, Dynamic Model Switcher, Dynamic-Macromodeling, ECL Compiler, ECO Compiler, EDAnavigator, Encore, Encore PQ, Evaccess, ExpressModel, Floorplan Manager, Formal Model Checker, FoundryModel, FPGA Compiler II, FPGA *Express*, Frame Compiler, Galaxy, Gatran, HANEX, HDL Advisor, HDL Compiler, Hercules, Hercules-Explorer, Hercules-II, Hierarchical Optimization Technology, High Performance Option, HotPlace, HSIM^{plus}, HSPICE-Link, i-Virtual Stepper, iN-Tandem, Integrator, Interactive Waveform Viewer, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, JVXtreme, Liberty, Libra-Passport, Libra-Visa, Library Compiler, Magellan, Mars, Mars-Rail, Mars-Xtalk, Medici, Metacapture, Metacircuit, Metamanager, Metamixsim, Milkyway, ModelSource, Module Compiler, MS-3200, MS-3400, Nova Product Family, Nova-ExploreRTL, Nova-Trans, Nova-VeriLint, Nova-VHDLlint, Optimum Silicon, Orion_ec, Parasitic View, Passport, Planet, Planet-PL, Planet-RTL, Polaris, Polaris-CBS, Polaris-MT, Power Compiler, PowerCODE, PowerGate, ProFPGA, ProGen, Prospector, Protocol Compiler, PSMGen, Raphael, Raphael-NES, RoadRunner, RTL Analyzer, Saturn, ScanBand, Schematic Compiler, Scirocco, Scirocco-i, Shadow Debugger, Silicon Blueprint, Silicon Early Access, SinglePass-SoC, Smart Extraction, SmartLicense, SmartModel Library, Softwire, Source-Level Design, Star, Star-DC, Star-MS, Star-MTB, Star-Power, Star-Rail, Star-RC, Star-RCXT, Star-Sim, Star-SimXT, Star-Time, Star-XP, SWIFT, Taurus, TimeSlice, TimeTracker, Timing Annotator, TopoPlace, TopoRoute, Trace-On-Demand, True-Hspice, TSUPREM-4, TymeWare, VCS Express, VCSi, Venus, Verification Portal, VFormal, VHDL Compiler, VHDL System Simulator, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (^{SM})

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
All other product or company names may be trademarks of their respective owners.

Printed in the U.S.A.

Formality Automated Setup File (SVF) Manual, version X-2005.12

Contents

About This Manual

This preface includes the following sections:

About This Manual

The *Formality Automated Setup File (SVF) Manual* provides information about and procedures for using the .svf file to assist in your design verification.

Related Publications

For additional information about <Product Name>, see

- Synopsys Online Documentation (SOLD), which is included with the software for CD users or is available to download through the Synopsys Electronic Software Transfer (EST) system
- Documentation on the Web, which is available through SolvNet at https://solvnet.synopsys.com/DocsOnWeb
- The documentation installed with the <Product Name> software and available through the <Product Name> Help menu
- The Synopsys MediaDocs Shop, from which you can order printed copies of Synopsys documents, at http://mediadocs.synopsys.com

Conventions

The following conventions are used in Synopsys documentation.

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

Customer Support

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

Accessing SolvNet

SolvNet includes an electronic knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. SolvNet also gives you access to a wide range of Synopsys online services including software downloads, documentation on the Web, and “Enter a Call to the Support Center.”

To access SolvNet,

1. Go to the SolvNet Web page at http://solvnet.synopsys.com.
2. If prompted, enter your user name and password. (If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.)

If you need help using SolvNet, click SolvNet Help in the Support Resources section.

Contacting the Synopsys Technical Support Center

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

- Open a call to your local support center from the Web by going to http://solvnet.synopsys.com (Synopsys user name and password required), then clicking “Enter a Call to the Support Center.”
- Send an e-mail message to your local support center.
  - E-mail support_center@synopsys.com from within North America.
  - Find other local support center e-mail addresses at http://www.synopsys.com/support/support_ctr.
- Telephone your local support center.
  - Call (800) 245-8005 from within the continental United States.
  - Call (650) 584-4200 from Canada.
  - Find other local support center telephone numbers at http://www.synopsys.com/support/support_ctr.

1

Introduction to the Automated Setup File

The automated setup file (.svf) helps Formality understand design changes caused by other tools used in the design flow. Formality uses this file to assist the compare point matching and verification process. This chapter includes the following sections:

Overview of the Automated Setup File

To use an automated setup file (file has the .svf extension), you first enable the creation of the file in the implementation tool. Alternately, you can manually create an automated setup file. Then you instruct Formality to read this file at the start of the verification process.

The benefit of an automated setup file is that it provides setup information to Formality automatically. It alleviates the need to enter setup information manually, a task that can be time consuming and error prone. For example, during synthesis a register might be duplicated to improve drive strength. This register duplication is recorded in the automated setup file. When Formality reads the .svf, it can account for the extra register during compare point matching and verification.

Creating an Automated Setup File

To create an automated setup file in Design Compiler, use the `set_svf` command. For example, use

```
dc_shell> set_svf myfile.svf
```

Whenever Design Compiler performs a design transformation of which Formality needs to be aware, the relevant guide command is added to the .svf file.

If you want to append the setup information to an existing setup file, use the following syntax.

```
dc_shell> set_svf -append myfile2.svf
```

Reading an Automated Setup File Into Formality

To read an automated setup file into Formality, use the `set_svf` command. You must read in the .svf file before you read in any design data (other than technology libraries).

The following example reads in the automated setup file, myfile.svf.

```
fm_shell(setup)> set_svf myfile.svf
SVF set to '/home/my/designs/myfile.svf'.
1
fm_shell(setup)>
```

The `set_svf` command returns the following:

- 0 for failure
- 1 for success

If you use `set_svf` without specifying the .svf file to use, Formality resets the automated setup file. However, the appropriate method for removing the stored setup data is to use the `remove_guidance` command.

Reading in Multiple Automated Setup Files

The automated setup file commands in the setup files describe transformations in an incremental fashion. The transformation occurs in the order in which the commands were applied as the RTL design was processed through design implementation or optimization. Therefore, the ability to read in multiple .svf files is important as no command in the file can be viewed completely independently because it describes the incremental transformation and is reliant on the context in which it is applied.

You can read multiple automated setup file files into Formality using the `read_svf` command. To order or instruct Formality to look for extensions other than .svf, use the `-ordered` or `-extension` options to the `set_svf` command, respectively. The ability to read in multiple setup files is useful when you have run bottom-up synthesis on your designs, generating multiple setup files.

You use the `-ordered` option to indicate that the list of setup files you specify are already ordered and should not be reordered by timestamp. If you use `-ordered` and list a directory or directories where the setup files are located, Formality might order the directory files in any order. The following example sets the order of two setup files, bot.svf and top.svf, for Formality to process:

```
set_svf -ordered bot.svf top.svf
```

The `-extension` option will load and automatically order all matching files in the directory you define, based the extension you define. For example, Formality automatically looks for files with the .svf extension. If you have automated setup files in a directory with extensions other than .svf, you use this option to instruct Formality to read and order those files with that extension. The ordering of the files is done using timestamp information typically found in the setup file header information. Formality doesn't require the timestamp information to be in the header, and can use specific guide commands for passing timestamp information directly. See the following section for information on the guide commands.

The following example instructs Formality to load and order setup files in the fmdir directory:

```
set_svf -extension fm fmdir
```

Setup File Commands

The .svf file consists of Tcl commands that start with `guide_`. The following is a list of these commands:

- `guide`
- `guide_architecture_db`
- `guide_architecture_netlist`
- `guide_change_names`
- `guide_datapath`
- `guide_fsm_reencoding`
- `guide_group`
- `guide_multiplier`
- `guide_reg_constant`
- `guide_reg_duplication`
- `guide_reg_encoding`
- `guide_reg_merging`
- `guide_transformation`
- `guide_ungroup`
- `guide_uniquify`
- `guide_ununiquify`

See Chapter 2, “Automated Setup Commands,” or the individual command’s online man page for specific information on each command.

You can also enter these commands manually. To do so, you must first enter guide mode by executing the `guide`command.

The following are the restrictions you need to follow when using the guide commands:

- None of the guide commands can trigger an immediate change to your designs when you apply them.
- You must issue the guide commands before you issue any design read commands. If you read in your design information and then run the `guide` command, Formality issues a warning message instructing you to remove the design information.

You can remove the setup file information that you entered using the guidance commands or through `set_svf` with the `remove_guidance` command. It is available in all modes of operation (guide, setup, and so on).

Writing a Text Version of the Automated Setup File in Formality

You can use the `report_guidance` command to generate a report of all automated setup file information that you entered using the guidance commands. The information is sent to the transcript, or optionally to a file that you specify using the `-to file_name` option. You can use the `-datapath` option to get a summary of the transformations in the automated setup file.

Use the following example to write out the automated setup file data to an unencrypted text file.

```
fm_shell > report_guidance
SVF hasn’t been set.
1

fm_shell> set_svf myfile.svf
SVF set to ‘myfile.svf’.
1

fm_shell> report_guidance -to myfile.txt
SVF set to ‘myfile.svf’.
1
```

If you are using the automated setup file to verify multipliers, the `report_guidance` command will also write out an unencrypted Verilog netlist to the directory ./fmsvf.

Using the Automated Setup File to Verify Multipliers

The arithmetic generator in Formality can create specific types of multipliers so the synthesized representation of your reference RTL more closely matches your gate implementation; thereby assisting with hard verification problems. This technique is particularly useful for the flat netlists where Formality cannot use the Data Path Solver (DPS) because it cannot identify the multipliers in the implementation.

For the traditional multiplier architectures supported by Design Compiler (CSA, NBW, and Wallace), the arithmetic generator can generate the appropriate multiplier architectures without any additional information. However, the more advanced multiplier

architectures from Design Compiler (mcarch and csmult) are so customized for each instance that the arithmetic generator needs additional information from it.

Using the automated setup file flow, you can instruct Design Compiler to automatically create netlist Verilog files for each multiplier in the design when performing synthesis. Design Compiler will produce a Verilog file containing the netlist for all multipliers as well as generate the automated setup file. Formality will verify each of these multipliers with the DPS; if the multipliers pass verification, Formality will load them into the reference design.

Along with the setup file, Design Compiler produces a directory (dwsvf_) containing a Verilog netlist. You must make both the .svf file and the dwsvf directory available for the arithmetic generator to perform a successful verification. See “Working With Multiplier Architectures” in the Formality User Guide for instructions on using the arithmetic generator in Formality.

You can view the contents of the Verilog netlist passed from Design Compiler to Formality with the `report_guidance` command. Using the `-to filename` option causes the architecture netlists found in the current setup file to be unencrypted. Formality copies the encrypted netlist files to filename.orig files and the unencrypted files overwrite the original netlists.

2

Automated Setup Commands

This chapter explains each of the commands available for use in the automated setup file. Examples are provided for each command. As mentioned in the previous chapter, Formality can use a automated setup file (.svf extension) to help decipher design changes caused by other tools used in the design flow. This chapter includes the following sections:

- guide_multiplier
- guide_reg_constant
- guide_reg_duplication
- guide_reg_encoding
- guide_reg_merging
- guide_transformation
- guide_ungroup
- guide_uniquify
- guide_ununiquify

guide

The `guide` command causes Formality to enter guide mode and enables all guide commands. Upon invocation, Formality starts in setup mode. After executing the `guide`, Formality switches to guide mode.

SVF commands can be executed only in guide mode. Guide mode cannot be entered after any design information (other than technology libraries) has been read. If the design has been read, it must be removed before guide mode can be entered.

The prompt changes to reflect the current mode, as shown below:

```
fm_shell (guide)>
fm_shell (setup)>
fm_shell (match)>
fm_shell (verify)>
```

The `guide` command returns 1 for success or 0 if the command fails.

Syntax

The syntax is as follows:

```
guide
```

Example

The following is an example of the `guide` command:

```
fm_shell (setup)> guide
1
fm_shell (guide)>
```

guide_architecture_db

The `guide_architecture_db` command associates a .db file with an architectural implementation.

You can specify the name of the .db file containing the associated architecture and the names of the library files referenced in the architectural implementation.

Syntax

The syntax is as follows:

```
guide_architecture_db
   [ -file filename ]
   [ libraries ]
```

Arguments

The arguments for the `guide_architecture_db` command are as follows:

`file` *`filename`*

Specifies the name of the .db file containing the associated architecture.

`libraries`

The .db files referenced in the architectural implementation.

Example

The following is an example of the `guide_architecture_db` command:

```
fm_shell (guide)> guide_architecture_db -file arch.db {
gtech }
1
```

guide_architecture_netlist

The `guide_architecture_netlist` command associates a netlist file with an architectural implementation.

You can specify the name of the netlist file containing the associated architecture and the names of the library files referenced in the architectural implementation.

Syntax

The syntax is as follows:

```
guide_architecture_netlist
   [ -file filename ]
   [ libraries ]
```

Arguments

The arguments for the `guide_architecture_netlist` command are as follows:

`file` *`filename`*

Specifies the name of the netlist file containing the associated architecture.

`libraries`

The .db files referenced in the architectural implementation.

Example

The following is a usage example of the `guide_architecture_netist` command:

```
fm_shell (guide)> guide_architecture_netlist -file arch.net
gtech.db
1
```

guide_change_names

The `guide_change_names` operation lists objects that have undergone a name change. It lists the name as it appears in the original design and the changed name as it appears in the modified implementation design.

Syntax

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

```
guide_change_names
   -design designName
   [ -instance instanceName ]
   [ changeBlock ]
```

Arguments

The arguments for the `guide_change_names` command are as follows:

`-design` *`designName`*

The design containing the names

`-instance` *`instanceName`*

The instance containing the names

`changeBlock`

A colon-separated list specifying the object type, the old name, and the new name

Example

The following is an example of the `guide_change_names` command:

```
guide_change_names \
   -design test \
   { cell:U1:mycell3 \
     cell:U2:mycell2 \
     cell:U3:mycell1 \
     port:data:myd \
     port:clock:myck \
     port:q:myq }
```

guide_datapath

The `guide_datapath` command identifies a datapath sub-design.

You can specify the names of the design and instance containing the datapath and the name of the file containing the datapath description (db or netlist format).

Syntax

The syntax is as follows:

```
guide_datapath
   -design designName
   [ -instance instanceName ]
   -body datapathBody
```

Arguments

The arguments for the `guide_datapath` command are as follows:

`-design` *designName*

The design containing the datapath

`-instance` *instanceName*

The instance containing the datapath

`-body` *filename*

The name of the file containing the datapath description (.db or netlist format)

Example

The following is an example of the `guide_datapath` command:

```
fm_shell (guide)> guide_datapath -design test -body dpath.db
1
```

guide_fsm_reencoding

You can use the `guide_fsm_reencoding` command to convey information about how a state machine should be re-encoded. The state machine re-encoding can involve a simple changing of the bit encoding, or it can involve changing the coding style (binary to one-hot, for example).

Syntax

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

```
guide_fsm_reencoding
   -design designName
   -previous_state_vector prevList
   -current_state_vector currList
   -state_reencoding stateList
```

Arguments

The arguments for the `guide_fsm_reencoding` command are as follows:

`-design` *`designName`*

The design containing the FSM

`-previous_state_vector` *`prevList`*

A list of the pre-reencoding register bits

`-current_state_vector` *`currList`*

A list of the post-reencoding register bits

`-state_reencoding` *`stateList`*

A list of the state reencodings

Example

The following is an example of the `guide_fsm_reencoding` command:

```
guide_fsm_reencoding \
   -design myfsm_0 \
   -previous_state_vector { out1_reg out0_reg } \
   -current_state_vector { Q4 Q3 Q2 Q1 } \
   -state_reencoding { begin:2#00:2#0001 \
      ok:2#01:2#0010 \
      nok:2#10:2#0100 \
      end:2#11:2#1000 }
```

guide_group

The `guide_group` command groups a set of instances. This will create an extra level of hierarchy. Use the guide_group operation to record the register name changes that result from the creation of the extra hierarchy. You can specify the current design context, name of the new design and instance, and old names and new names of register instances as options to the guide_group operation.

Syntax

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

```
guide_group
   -design designName
   [ -instance instanceName ]
   [ -cells cellList ]
   -new_design newDesignName
   -new_instance newInstanceName
  [ groupBlock ]
```

Arguments

The arguments for the `guide_group` command are as follows:

`-design` *`designName`*

The name of the design containing the objects

`-instance` *`instanceName`*

The name of the instance containing the objects

`-cells` *`cellList`*

A list of the cells being grouped

`-new_design` *newDesignName*

The name of the new design being created

`-new_instance` *newInstanceName*

The name of the instance of the new design being created

groupBlock

A colon-separated list of old and new cell names

Example

The following is an example of the `guide_group` command:

```
guide_group \
   -design shift8 \
   -cells { shift4_i_1 \
            shift4_i_2 \
            shift_4_i_2 \
            shift_4_i_3 } \
   -new_design foo \
   -new_instance foo_i_1 \
   { shift4_i_1:foo_i_1/shift4_i_1 \
     shift4_i_2:foo_i_1/shift4_i_2 }
```

guide_multiplier

The `guide_multiplier` command identifies a sub-design as a multiplier with a specific architecture.

You can specify the names of the design and instance containing the multiplier, the architecture of the multiplier (csa, nbw, wall, csmult or mcarch), and the name of the file containing the multiplier description (.db or netlist format).

Syntax

The syntax is as follows:

```
guide_multiplier
   -design designName
   [ -instance instanceName ]
   -arch arch
   -body fileName
```

Arguments

The arguments for the `guide_multiplier` command are as follows:

`-design` *designName*

The design containing the multiplier

`-instance` *instanceName*

The instance containing the multiplier

`-arch` *`arch`*

The architecture of the multiplier: csa, nbw, wall, csmult or mcarch

`-body` *`filename`*

The name of the file containing the multiplier description (.db or netlist format)

Example

The following is an example of the `guide_multiplier` command:

```
fm_shell (guide)> guide_multiplier \
   -design fei_int \
   -instance mul_24/mult/mult \
   -arch mcarch \
   -body fei_int_DW02_mult_8_8_0
1
```

guide_reg_constant

The `guide_reg_constant` command records that a register was optimized to a constant value by another tool in the design flow. You specify the constant value as an argument to the guide_reg_constant operation.

Syntax

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

```
guide_reg_constant
   [ -design designName ]
   instanceName
   constantVal
```

Arguments

The arguments for the `guide_reg_constant` command are as follows:

`-design` *`designName`*

The name of the design containing the constant registers (only the design name is required; the workspace and container are not required)

`instanceName`

The hierarchical path from the specified design to the target constant register

`constantVal`

The constant value

Examples

The following are some examples of the `guide_reg_constant` command.

To set the instance U1 inside design to 1:

```
guide_reg_constant -design test U1 1
```

To set the instance r:WORK/top/mid_inst_0/bot_inst_0/state[0] to constant 0:

```
guide_reg_constant -design top mid_inst_0/bot_inst_0/
state[0] 0
```

guide_reg_duplication

The `guide_reg_duplication` command records that a single register in the reference design was duplicated in the implementation design. This can occur if the synthesis tool determines that a register has too many loads. You can duplicate the register so that each of the resulting registers drives fewer loads.

Syntax

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

```
guide_reg_duplication
   [ -design designName ]
   -from fromReg
   -to toList
```

Arguments

The arguments for the `guide_reg_duplication` command are as follows:

`-design` *`designName`*

The name of the design containing the duplicate registers

`-from` *`fromReg`*

The original register

`-to` *`toList`*

A list of the duplicated registers

Example

The following is an example of the `guide_reg_duplication` command:

```
guide_reg_duplication -design test -from U1 -to { U1 U2 U3 }
```

guide_reg_encoding

The `guide_reg_encoding` command identifies registers whose encoding has changed (usually from binary to carry-save). For a binary to carry-save encoding, the first bit on each line represents the binary value while the remaining bits represent the carry save encoding.

You can specify the name of the design containing the encoded register, the original style of the register (binary, CS2, and so on), the new style of the register (binary, CS2, and so on), and the register bits that have changed.

Syntax

The syntax is as follows:

```
guide_reg_encoding
   -design designName
   -from fromStyle
   -to toStyle
   {bit:bit:[:bit]*}
```

Arguments

The arguments for the `guide_reg_encoding` command are as follows:

`-design` *`designName`*

The name of the design containing the duplicate registers

`-from` *`fromStyle`*

The original style of the register (binary, CS2, and so on)

`-to` *`toStyle`*

The new style of the register (binary, CS2, and so on)

{*`bit:bit:[:bit]*`*}

The register bits that have changed

Example

The following is an example of the `guide_reg_encoding` command:

```
fm_shell (guide)> guide_reg_encoding \
   -design test \
   -from binary \
   -to CS2 \
   R[0]:R_sum[0]:R_carry[0] \
   R[1]:R_sum[1]:R_carry[1]
1
```

guide_reg_merging

The `guide_reg_merging` command records that multiple registers in the reference design were merged into one register in the implementation design. This can occur if the synthesis tool determines that two or more registers always have the same state.

Syntax

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

```
guide_reg_merging
   [ -design designName ]
   -from fromList
   -to toReg
```

Arguments

The arguments for the guide_reg_merging command are as follows:

`-design` *designName*

The name of the design containing the merged registers

`-from` *fromList*

A list of the merged registers

`-to` *toReg*

The final register

Example

The following is an example of the `guide_reg_merging` command:

```
guide_reg_merging \
   -design top \
   -from { Q2 Q3 } \
   -to Q2
```

guide_transformation

The guide_transformation command identifies registers which are duplicates of each other.

You can specify the name of the design containing the duplicate registers, the type of transformation, (share, tree, map or merge), and the following lists:

- Transformation inputs
- Transformation outputs
- Transformation control signals
- Transformation virtual signals
- Resources in the transformation pre-graph
- Assignments in the transformation pre-graph
- Resources in the transformation post-graph
- Assignments in the transformation post-graph
- Datapath elements in the transformation

Syntax

The syntax is as follows:

```
guide_transformation
   -design designName
   -type type
   -input inputList
   -output outputList
   [ -control controlList ]
   [ -virtual virtualList ]
   [ -pre_resource preResourceList ]
   [ -pre_assign preAssignList ]
   [ -post_resource postResourceList ]
   [ -post_assign postAssignList ]
   [ -datapath datapathList ]
```

Arguments

The arguments for the `guide_transformation` command are as follows:

`-design` *designName*

The name of the design containing the transformations

`-type` *type*

The type of transformation: share, tree, map, or merge

`-input` *inputList*

A list of the transformation inputs

`-output` *outputList*

A list of the transformation outputs

`-control` *controlList*

A list of the transformation control signals

`-virtual` *`virtualList`*

A list of the transformation virtual signals

`-pre_resource` *`preResourceList`*

A list of the resources in the transformation pre-graph

`-pre_assign` *`preAssignList`*

A list of the assignments in the transformation pre-graph

`-post_resource` *`postResourceList`*

A list of the resources in the transformation post-graph

`-post_assign` *`postAssigntList`*

A list of the assignments in the transformation post-graph

`-datapath` *`datapathList`*

A list of datapath elements in the transformation

Example

The following is an example of the `guide_transformation` command:

```
fm_shell (guide)> guide_transformation
   -design general_tree \
   -type tree \
   -input { 4 src1 4 src2 4 src4 } \
   -output { 6 O1 } \
   -control { ctrl1 = cond ctrl2 } \
   -pre_resource { { 5 5 } add_7 = DIV { { src1 2 4 } { src2
2 3 ZERO 3 } } } \
   pre_resource { { 5 5 } add_8 = DIV { { src1 2 4 } { src2
2 4 ZERO 5 } } } \
   -pre_resource { { 5 5 } add_9 = DIV { { src1 ZERO 5 } {
src2 ZERO 5 } } } \
   -pre_resource { { 6 4 4 } sub_9 = USUB { { add_9 ZERO 6 } \
      { src4 ZERO 6 } } } \
   -pre_assign { O1 = { sub_9 } } \
   -post_resource { { 6 7 4 3 } sub_1_root_sub_9 = USUB \
      { { src1 ZERO 6 } { src4 ZERO 6 } } } \
   -post_resource { { 6 7 3 2 5 } add_0_root_sub_9 = UADD \
      { { src2 ZERO 6 } { sub_1_root_sub_9 } } } \
   -post_assign { O1 = { add_0_root_sub_9 } }
1
```

guide_ungroup

The `guide_ungroup` command removes a level or levels of hierarchy. Also, you use this command to specify how registers names are to be changed when the hierarchy is removed. You can specify the current design context, names of the removed hierarchical blocks, and old names and new names of compare points as options to the `guide_ungroup` command.

Syntax

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

```
guide_ungroup
   -design designName
   [ -instance instanceName ]
   [ -cells cellList ]
   [ ungroupBlock ]
```

Arguments

The arguments for the `guide_ungroup` command are as follows:

`-design` *designName*

The name of the design containing the objects

DC ungroup options.

`-instance` *instanceName*

The name of the instance containing the objects

`-cells` *cellList*

A list of the cells being ungrouped

ungroupBlock

A colon-separated list of old cell:new cell pairs

Example

The following is an example of the `guide_ungroup` command:

```
guide_ungroup \
   -design shift8 \
   -instance shift16/shift8_i_1 \
   -cells { foo } \
   -flatten \
   { shift4_i_1/shift2_i_1/dout_reg:dout_reg \
     shift4_i_1/shift2_i_1/state1_reg:state1_reg \
     shift4_i_1/shift2_i_2/dout_reg:dout_reg1 \
     shift4_i_1/shift2_i_2/state1_reg:state1_reg1 \
     shift4_i_2/shift2_i_2/state1_reg:state1_reg3 }
```

guide_uniquify

The `guide_uniquify` operation provides information about components of the design that have been uniquified. When a design is uniquified, all instances of a component instantiated multiple times are assigned unique names and thus become unique components. The V-SDC file will contain:

- The current design where the uniquify is to take place
- The new design names that are to be created for each uniquified instance

Note that changing the design name does not necessarily change the instance specific names of the registers in the design. This is because design names don't appear in the instance specific pathnames of objects. However, certain combinations of this command and the hierarchy-modifying commands `guide_group` and `guide_ungroup` can change the names of registers and hierarchical blocks.

Syntax

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

```
guide_uniquify
   -design designName
   [ uniquifyBlock ]
```

Arguments

The arguments for the `guide_uniquify` command are as follows:

`-design` *designName*

The name of the design containing the objects being uniquified

uniquifyBlock

A colon-separated list of old cell:new cell pairs

Example

The following is an example of the `guide_uniquify` command:

```
guide_uniquify \
   -design shift4 \
   { shift2_i_1:shift2_0 \
     shift2_i_2:shift2_1 }
```

guide_ununiquify

The guide_ununiquify operation reverses the changes made to the design by the guide_uniquify command. When a design is ununiquified, all instances of a design that had been uniquified are changed back to non-unique designs. The V-SDC file will contain:

- The current design where the ununiquify is to take place
- The new design names that are to be created for each uniquified instance

Note that changing the design name does not necessarily change the instance specific names of the registers in the design. This is because design names don't appear in the instance specific pathnames of objects. However, certain combinations of this command and the hierarchy-modifying command guide_group and guide_ungroup can change the names of registers and hierarchical blocks.

Syntax

The syntax for the guide_ununiquify operation is as follows:

```
guide_ununiquify
   -design designName
   [ ununiquifyBlock ]
```

Arguments

The arguments for the `guide_ununiquify` command are as follows:

`-design` *`designName`*

The name of the design containing the objects being folded

`uniquifyBlock`

A colon-separated list of old cell:new cell pairs

Example

The following is an example of the `guide_ununiquify` command:

```
guide_ununiquify \
   -design top \
   { U2:mid \
     U1:mid \
     U2/U3:bot \
     U1/U3:bot }
```