SIEMENS EDA

Sequential Logic Equivalence Checker (SLEC) Reference Manual

Software Version 2023.2

SIEMENS

About Siemens Digital Industries Software

Siemens Digital Industries Software is a leading global provider of product life cycle management (PLM) software and services with 7 million licensed seats and 71,000 customers worldwide. Headquartered in Plano, Texas, Siemens Digital Industries Software works collaboratively with companies to deliver open solutions that help them turn more ideas into successful products. For more information on Siemens Digital Industries Software products and services, visit www.siemens.com/plm.

Support Center: support.sw.siemens.com
Send Feedback on Documentation: support.sw.siemens.com/doc_feedback_form

Table of Contents

Chapter 1
Introduction

This reference manual describes the Sequential Logic Equivalence Checker (SLEC) from Siemens EDA.

Syntax Conventions

This manual uses the following command line syntax conventions.

Table 1. Conventions for Command Line Syntax

Convention	Example	Usage
Boldface	create_clock **-name *name* -period *period***	A boldface font indicates a required argument.
[ ]	create_one [-spec | -impl]	Square brackets enclose optional arguments. Do not enter the brackets.
Italic	create_and [-name *inst_name*] ***signal_1 signal_2*** ...	An italic font indicates a user-supplied argument.
{ }	read_reset_image { **-spec** | **-impl** } **-file *file* -time *time*** [-scope *scope*] [-outfile *outfile*] [-verbose]	Braces enclose arguments to show grouping. Do not enter the braces.
|	create_zero [-spec | -impl]	The vertical bar indicates an either/or choice between items. Do not include the bar in the command.
Underline	view_waveform [-spec | -impl] [cex | setup | tb]	An underlined item indicates either the default argument or the default value of an argument.
...	findlist [-input | -output | -clock] [-user] [-hier] [-require *require* ...] [-append *append* ...] [*expr*]	An ellipsis follows an argument that may appear more than once. Do not include the ellipsis when entering commands.

Chapter 2
Command Reference

This chapter describes the commands available in the SLEC tool.

assert_all_maps_proven

Checks whether all maps were proven in verification.

Usage

```
assert_all_maps_proven
```

Arguments

None.

Description

This command parses various log files, including *slec.log* and *results.log* to determine whether all maps were proven. Execute this command after the verify command. If all maps were proven during verification, the command returns zero. Otherwise, it reports an error and exits with a non-zero return code.

assume_properties

Assumes that the specified properties always hold true. This command can only be called after the build_design command.

Usage

assume_properties **-ovl** | **-prop** | **-sva**
[-spec | -impl | -list { *tcl_list_of_ovl_instances* | *tcl_list_of_properties* | *tcl_list_of_sva_properties* }]
[-assert_always | -assert_one_hot | -assert_never | -abr | -abw | -asc | -cas | -dbz | -ise | -umr | -assume]
[-induct *induction_depth*]

Arguments

- **-ovl** | **-prop** | **-sva**

  Specifies the type of check to be assumed true.

  Specify -ovl to assume all OVL properties are always true.

  Specify -prop to assume all user assert statements are always true. Note that this option requires the verification mode to be set to "property_checks".

  Specify -sva to assume all user System Verilog assume SVA statements are always true.

- -spec | -impl | -list { *tcl_list_of_ovl_instances* | *tcl_list_of_properties* | *tcl_list_of_sva_properties* }

  Specify -spec to assume that all properties in the specification design are always true.

  Specify -impl to assume that all properties in the implementation design are always true.

  Specify -list to provide a list of OVL properties, user-defined properties, or SVA properties that should be assumed true during verification.

  When none of these options are specified, it is assumed that all properties in the specification and implementation design are true.

  Wildcards are supported. Wildcards extend UNIX shell * and ? glob matching with a ** operator, which matches zero or more levels of hierarchy. The wildcard matching algorithm follows the semantics of the find command.

- -assert_always | -assert_one_hot | -assert_never | -abr | -abw | -asc | -cas | -dbz | -ise | -umr | -assume

  Assumes all properties of the type specified are true.

  Specify -assert_always to assume that properties specified by OVL instances of type assert_always are always true. Note that the -ovl option is required with this option.

  Specify -assert_one_hot to assume that properties specified by OVL instances of type assert_one_hot are always true. Note that the -ovl option is required with this option.

  Specify -assert_never to assume that properties specified by OVL instances of type assert_never are always true. Note that the -ovl option is required with this option.

  Specify -abr to assume that there are no out-of-bounds array and bit-vector reads in the design. Note that the -prop option is required with this option.

Specify -abw to assume that there are no out-of-bounds array and bit-vector writes in the design. Note that the -prop option is required with this option.

Specify -asc to assume all user assert statements are always true. Note that the -prop option is required with this option.

Specify -cas to assume that there are no incomplete case statements in the design. Note that the -prop option is required with this option.

Specify -dbz to assume that there are no divide-by-zero and modulo-by-zero operations in the design. Note that the -prop option is required with this option.

Specify -ise to assume that there are no illegal shift operations in the design. Note that the -prop option is required with this option.

Specify -umr to assume that there are no uninitialized memory reads in the design. Note that the -prop option is required with this option.

Specify -assume to assume that System Verilog assume SVA properties are always true.

- -induct *induction_depth*

  Specifies the induction depth.

Description

The assume_properties command tells SLEC to assume that the specified properties always hold true. When no option is specified, all properties in the specification and implementation design are assumed to be true. Note that in case the designs are proven equivalent, SLEC generates a conditional proof instead of a full proof. This is because SLEC has not validated the values. Instead, SLEC has assumed that the specified properties always hold true.

Examples

Example 1: Specifies that SLEC verify the implementation design under the assumption that the properties specified by OVL instances of the type assert_always are always true.

```
assume_properties -ovl -impl -assert_always
```

Example 2: Specifies that SLEC verify the specification design under the assumption that the user assert statements prop_user_assert_ln6 and prop_user_assert_ln10 are always true.

```
assume_properties -prop -list {spec.prop_asc_ln6 spec.prop_asc_ln1}
```

Example 3: Specifies that SLEC verify the implementation design under the assumption that the properties specified by the OVL instances impl.assert_always_en and impl.abc.assert_one_hot_sel are always true.

```
assume_properties -ovl -list { {impl.assert_always_en} \
 {impl.abc.assert_one_hot_sel} }
```

Example 4: Specifies that SLEC verify the design under the assumption that all user assert statements in the specification design always hold true.

```
assume_properties -prop -spec
```

Example 5: Specifies that SLEC verify the design under the assumption that there are no out-of-bounds array and bit vector reads in the specification and implementation designs.

```
assume_properties -spec -prop -abr
assume_properties -impl -prop -abr
```

Example 6: Specifies that SLEC verify the design under the assumption that the System Verilog SVA assume properties in the specification and implementation designs always hold true.

```
assume_properties -spec -sva -assume
assume_properties -impl -sva -assume
```

build_db

Builds a database representing the design library.

Usage

build_db [-spec | -impl | -cons]

Arguments

- -spec | -impl | -cons

  Specifies the design library to build.

Description

The build_db command builds a database using the linked and elaborated design. For example, specify the following command to build the specification design library.

```
build_db -spec
```

After building the specification or implementation library, an attempt is made to identify one module in the library which is not instantiated in any other modules. If only one such module is found, it is considered to be the top module for that design library. If multiple top modules are found, then the first one read in is marked as the top module. An error will be issued if the design library has not been linked or has already been built.

Examples

Example 1: This example illustrates how to read a SystemC design into the specification library, link the design and then use the build_db command to build the database.

```
read_design -spec design.cpp
link_design -spec
build_db -spec
```

Example 2: This example shows how to read multiple VHDL files into different VHDL libraries in the SLEC implementation library, and then subsequently link and build those files.

```
read_design -impl -library LIB1 entity1.vhdl
read_design -impl -library LIB2 entity2.vhd
read_design -impl top_entity.vhd
link_design -impl
build_db -impl
```

build_design

Reads a set of design files into a design library, links them, and builds a design database. Returns a list of top-level module names in the synthesized design.

Usage

build_design **-spec** | **-impl** | **-cons** [-vlog | -vhdl | -systemc | -svlog] [-top *top_module*] ***arguments***

Arguments

- **-spec** | **-impl** | **-cons**

  Adds modules to the specification, implementation, or constraints design library.

- -vlog | -vhdl | -systemc | -svlog

  Specifies the design language of the files being read in. If omitted, the suffix of the first file read in is used to establish the design language (Verilog if *.v,* VHDL if *.vhd**, SystemC if *.h** or *.c**, and SystemVerilog if *.sv*).

- -top *top_module*

  Specifies the top-level module.

Verilog and SystemVerilog Arguments

- +define+*macro* [=*macro_body*]

  Defines a macro.

- -f *list_file*

  Uses a list file, which contains a set of files and options to be read in.

- -y *dir_name*

  Searches for unresolved modules in the directory <*dir_name*>.

- -v *file_name*

  Searches for unresolved modules in the file <*file_name*>.

- -cpu_stats

  Displays CPU usage statistics.

- -defparam *list_of_parameter-value_pairs*

  Specifies the list of parameter value pairs to override default values for the parameters of the top-level module.

  For example, to override the values of parameters p1 and WIDTH in the top module of the specification design:

  ```
  build_design -spec -defparam "{p1 1} {WIDTH $width}" cpu.v
  ```

- -mem_stats

  Displays memory usage statistics.

- +libext+*ext*

Searches for unresolved modules in a library directory using extension <*ext*>.

- +incdir+*dir* ...

  Searches + separated directories to resolve `include directives.

- {-svlog_2005 | -svlog_2009 | -svlog_2012 | -svlog_2017}

  Overrides the default SystemVerilog version set with the system_verilog_version global.

- {-vlog_1995 | -vlog_2001 | -vlog_2005}

  Overrides the default Verilog version set with the verilog_version global.

- ***filenames*** ...

  A whitespace separated list of design files to be read into SLEC which may include absolute or relative directory paths. By default, any additional file references included within the design files are searched for in the current working directory, and additional include paths may be included using +incdir+<*dir*> options.

- -sfcu

  Single File Compilation Unit.

  Treats each file specified in a compilation command line as a separate compilation unit. The types, functions, tasks, and so on defined in one compilation unit are not visible to the other compilation units.

- -mfcu

  Multiple File Compilation Unit.

  Treats the files specified in a compilation command line as a single compilation unit. This switch puts both Verilog and SystemVerilog files in a single compilation unit. This is the default compilation behavior.

VHDL Arguments

- -library *library*

  Specifies that the VHDL units being read in belong to the VHDL logical library <*library*>.

- -defparam *list_of_generic-value_pairs*

  Specifies the list of generic value pairs to override default values for the generics of the top-level entity.

  For example, to override the values of generics p1 and WIDTH in the top-entity of the specification design:

  ```
  build_design -spec -defparam "{p1 1} {WIDTH $width}" cpu.vhd
  ```

- {-vhdl_1987 | -vhdl_1993 | -vhdl_2000 | -vhdl_2002 | -vhdl_2008}

  Overrides the default VHDL version set with the vhdl_version global.

- ***filenames*** ...

  A whitespace separated list of design files to be read into SLEC which may include absolute or relative directory paths. By default, any additional file references included within the design files are searched for in the current working directory.

C++ and SystemC Arguments

- -I*include-dir*

  Appends a directory to the include file search path.

- -std=c++98 | -std=c++11

  Specifies the C++ language version.

  - c++98 : use default C++ version.
  - c++11 : enables support for C++11 constructs.

Note:
SLEC compiles C code, which may be located in **.c* files, as C++ code.

- -D*macro*, -D*macro*=*value*

  Defines a macro.

- -w

  Suppresses common warnings.

- ***filenames*** ...

  A whitespace separated list of design files to be read into SLEC which may include absolute or relative directory paths. By default, any additional file references included within the design files are searched for in the current working directory, and additional include paths may be included using -I<*include-dir*> options.

Other options, such as -U<*macro*>, are not understood by the reader.

The include path for *systemc.h* is automatically included and *should not be specified*.

Description

The build_design command is an alias for reading in a design, linking it, and building the design database. For example,

```
build_design -spec design_modules.v
```

is equivalent to:

```
read_design -spec design_modules.v
link_design -spec
build_db -spec
```

Likewise, the following command reads a SystemC design and builds a database in the specification library.

```
build_design -spec design.cpp
```

For SystemVerilog, the design file can use a *.v* or *.sv* extension. In case the file uses a *.v* extension, the -svlog option needs to be explicitly specified to build the design. In this example, build_design reads in a SystemVerilog file and builds a database in the specification library.

```
build_design -svlog -spec design.v
```

In this example, a Verilog design is read into the specification library and a VHDL design is read into the implementation library. In such a scenario, all entities are created in the default VHDL work library. Refer to the read_design command to understand what happens when multiple VHDL libraries are used.

```
build_design -spec design1.v design2.v ./sub_dir/design3.v
build_design -impl design1.vhd design2.vhdl ./sub_dir/design3.vhdl
```

When reading transactors into the constraints library, use a separate build_design command for each language. For example,

```
build_design -cons -top mod1 trans1.cpp
build_design -cons -top mod2 trans2.v
build_design -cons -top mod3 trans3.vhdl
```

The language version options for Verilog, SystemVerilog, and VHDL designs are determined by the verilog_version, system_verilog_version, and vhdl_version globals respectively. These typically default to the latest version available.

The build_design command returns the top-level module or modules in the design. For the spec and impl libraries, build_design returns a single top-level module. If SLEC cannot determine the top-level module, it returns an error.

For the constraints library, build_design may return multiple top-level modules if -top is not specified. To avoid confusion, specify the -top argument when building the constraints library.

case_split

Splits a verification run into multiple smaller runs to improve verification runtime. SLEC can perform case splits internally or externally. For internal case split, SLEC solves all cases within the same verification run. For external case split, SLEC spawns a separate subsession for each case.

Usage

(Internal): case_split ***signal_list***

(External): case_split [-nodefault] { **-external** ***input_signal*** } *values*

Arguments

- ***signal_list***

  (Internal case split only): A required input parameter specifying a list of signals to enumerate for the case split. This parameter must be a complete signal and not a slice of signals.

  Note:
  You can use wildcards to specify the list of signals. For example:

  ```
  case_split {spec.*_signal*}
  ```

- -nodefault

  (External case split only): An optional argument that instructs SLEC to test only the specified values, without testing a scenario for all other possible values (see the Description section below). If this argument is not specified, the constant or waveform groups specified in the values argument must not overlap. Furthermore the constraint waveforms should not include a '+' operator to extend the sequences.

  Note:
  The -nodefault argument must precede the -external argument.

- **-external**

  A required argument that specifies to perform an external case split, solving each case in a separate subsession.

  Note:
  Although running verification with external case split can reduce verification runtime, it does not include certain scenarios such as interleavings or specific reset state conditions. As such, verification with external case split is less comprehensive than with internal case split or leaving case split off altogether.

- ***input_signal***

  (External case split only): A required input parameter specifying a design input signal to enumerate for the case split. This must be a single input signal or slice of an input signal.

- *values*

  (External case split only): An optional input parameter specifying a list of values or explicit waveforms for the input signal. If the input is 8-bit or smaller and no value is specified, SLEC enumerates all possible values. If the input is larger than 8-bit and no value is specified, SLEC ignores the command and issues a warning.

Description

The case_split command splits a verification run into multiple smaller runs to improve verification runtime. SLEC can perform case splits internally or externally. For internal case split, SLEC solves each case within the same verification run. For external case split, SLEC spawns a separate subsession for each case.

For a detailed explanation of capacity and runtime issues, along with strategies for mitigation, see the *"Optimizing SLEC Capacity and Runtime"* Application Note in the *<product_version>/slec/doc/app_notes* directory.

Internal Case Split

Internal case splits serve as hints for the solvers on how to divide a large verification run into smaller cases to improve performance. The *signal_list* input parameter must be four bits or fewer; otherwise, the number of cases may be too large. This parameter must be a complete signal and not a slice of signals.

External Case Split

External case splits divide a verification run into multiple smaller runs to improve verification runtime and spawns a separate subsession for each case.

By default, SLEC spawns synchronous subsessions, with each verification run occurring sequentially. To spawn asynchronous subsessions, which can be processed simultaneously, set the external_case_split_async_subsessions global to true. To distribute asynchronous subsessions across multiple machines, configure SLEC as described in Distributed Processing in SLEC. If remote workers are not configured, SLEC, by default, uses one local asynchronous worker, and the performance is similar to using synchronous (blocking) subsessions. Verification reports success only if all subsessions succeed.

If you do not specify the -nodefault argument, SLEC tests an additional case where the specified input signal does not match any of the specified values. If, for example, three values are listed for the input signal, a fourth subsession tests the case for all other possible values. If you wish to only run verification for the specified values, include the -nodefault argument with the command.

When using the external case split feature in SLEC, the following restrictions apply:

- No other constraints are allowed on the input signals being case split.
- Only one case_split command is allowed per input signal.
- The throughput of the design whose input is being case split must be one. This requirement does not apply to the other design not being case split.
- The constraints are not applied during reset.

The detailed verification results and formal coverage results are not merged from subsessions into the parent SLEC process. Rather, each subsession reports a simple pass/fail result.

When running external case split, the parent SLEC process launches multiple subsessions, each with its own child SLEC process. The child subsessions run the same script as the parent, but with different

parameters. To determine whether the current session is a parent session or child subsession, you can run the following procedure:

```
ecs::is_child
```

The ecs::is_child procedure indicates whether the current session is a child subsession. A result of 0 indicates the current session is the parent session. A result of 1 indicates the current session is a child subsession.

Each child subsession runs with a unique set of parameters based on the input values specified. You can access the values corresponding with the current child subsession with the following procedure:

```
ecs::get_combination
```

The ecs::get_combination procedure returns a TCL list of integer indices indicating the parameters designated for the current subsession. The indices are ordered in the same manner in which they appear in the script. The value of each index ranges from 0 to N-1, where N is the total number of possible values for the signal.

check_properties

Checks if the specified properties are always true. This command can only be called after the build_design command.

Usage

check_properties **-ovl** | **-prop** | **-sva**
[-spec | -impl | -list { *tcl_list_of_ovl_instances* | *tcl_list_of_sva_properties* | *tcl_list_of_properties* }]
[-assert_always | -assert_one_hot | -assert_never | -abr | -abw | -asc | -cas | -dbz | -ise | -umr | -assert]
[-induct *induction_depth*]

Arguments

- **-ovl** | **-prop** | **-sva**

  Specifies whether the checks are to be performed for OVL properties or user-defined properties.

  Specify -ovl to check if the specified OVL properties are always true.

  Specify -prop to check if the specified user assert statements are always true. Note that this option requires the verification mode to be set to property_checks.

  Specify -sva to check if the specified user System Verilog SVA assert statements are always true.

- -spec | -impl | -list { *tcl_list_of_ovl_instances* | tcl_list_of_sva_properties | *tcl_list_of_properties* }

  Specifies whether the checks are to be performed for properties in the specification design or implementation design. Use the -list option to perform checks on a specific list of properties.

  When no option is specified, SLEC checks if all properties in the specification and implementation design are always true.

  Wildcards are supported. Wildcards extend UNIX shell * and ? glob matching with a ** operator, which matches zero or more levels of hierarchy. The wildcard matching algorithm follows the semantics of the find command.

- -assert_always | -assert_one_hot | -assert_never | -abr | -abw | -asc | -cas | -dbz | -ise | -umr | -assert

  Use one of these options to filter the list of properties checked. Note that the -assert_always, -assert_never and -assert_one_hot options require the -ovl option to be specified. For all other options, the -prop option must be specified.

  Specify -assert_always to verify that the properties specified by OVL instances of the type assert_always are always true.

  Specify -assert_never to verify that the properties specified by OVL instances of the type assert_never are always true.

  Specify -assert_one_hot to verify that the properties specified by OVL instances of the type assert_one_hot are always true.

  Specify -abr to check the design for out-of-bounds array and bit-vector reads.

  Specify -abw to check the design for out-of-bounds array and bit-vector writes.

Specify -asc to verify the design after checking that the properties specified using the user assert statements are always true.

Specify -cas to check the design for incomplete case statements.

Specify -dbz to check the design for divide-by-zero and modulo-by-zero operations.

Specify -ise to check the design for illegal shift operations.

- The following checks are performed for excessive shifting:
  - Left-shift (<<) and right-shift (>>) operators
  - shift_left and shift_right functions defined in the file *mgc_sc_math.h*
- The following data types are checked:
  - Integral C data types (int, char, short, long, unsigned, and so on)
  - ac_int<>, ac_fixed<>
  - sc_* data types
- The following are supported data types and their legal shift values:
  - unsigned char : (>= 0) && (< 32)
  - unsigned int : (>= 0) && (< 32)
  - unsigned long long : (>= 0) && (<64)
  - ac_int : Defined for all shift amounts
  - ac_fixed : Defined for all shift amounts
  - sc_(u)int<W> : (>= 0) && (< 64)
  - sc_bv, sc_lv, sc_big(u)int : (>=0)
  - sc_fixed : (>=0)

Specify -umr to check the design for uninitialized memory reads.

Specify -assert to verify that the System Verilog assert SVA properties are always false.

- -induct *induction_depth*

  Specifies the induction depth.

Description

Checks if the properties specified are true.

Note:
To generate testbenches for OVL instances, set the ovl_library_path global to point to the directory containing the simulation models for the OVL assertions.

Examples

Example 1: Checks that the properties specified by the OVL instances of type assert_always in the implementation design are always true.

```
check_properties -ovl -impl -assert_always
```

Example 2: Checks that the user assert statements prop_user_assert_ln6 and prop_user_assert_ln10 in the specification design are always true.

```
check_properties -prop -list {spec.prop_asc_ln6 spec.prop_asc_ln1}
```

Example 3: Checks that the properties specified by the OVL instances impl.assert_always_en and impl.abc.assert_one_hot_sel in the implementation design are always true.

```
check_properties -ovl -list { {impl.assert_always_en}
 {impl.abc.assert_one_hot_sel} }
```

Example 4: Checks that all user assert statements in the specification design always hold true.

```
check_properties -prop -spec
```

Example 5: Checks that there are no out-of-bounds array and bit-vector reads in the specification design.

```
check_properties -prop -spec -abr
```

compare_version

Compares the specified version with the current SLEC version.

Usage

compare_version ***version_num***

Arguments

- ***version_num***

  The version number with which to compare. Must be a properly formatted SLEC version corresponding to a 10.X release or later.

Description

The compare_version command compares the specified version with the current SLEC version. It returns the following values:

- -1: The specified version is less than the current version.
- 0: The specified version is equal to the current version.
- 1: The specified version is greater than the current version.

The ordering is based upon version numbers and not release date. For example, version 10.4b is greater than 10.4a_2, even if 10.4b was released first.

config_coverage_scope

Filters modules or instances from the formal coverage analysis and reporting.

Usage

config_coverage_scope [-include | -exclude] {**-spec** | **-impl**}
[-module *module_list* | -inst *inst_list* | all]

Arguments

- -include | -exclude

  Specifies to include or exclude specified instances or modules from the coverage report. By default, both the specification and implementation designs are included.

- **-spec** | **-impl**

  Specifies whether the modules or instances are from the specification or implementation design.

- -module *module_list* | -inst *inst_list* | all

  Specifies a list of modules or instances. Use the all argument if the entire specification or implementation design needs to be included or excluded.

Description

The config_coverage_scope command filters out modules or instances on which the coverage information is (or is not) required. By default, the coverage report is generated for all the applicable instances in the specification and implementation designs. Use the -exclude and -include arguments to exclude or include modules or instances from the coverage report. The specification design is always included, unless it is explicitly excluded using the all argument.

You can specify multiple instances of the config_coverage_scope command. In case of conflicts, subsequent commands override earlier commands. See the Examples section.

You must first run the build_design command on the corresponding design before executing the config_coverage_scope command.

For further details, refer to the application note *SLEC Coverage* located in the *<product_version>/slec/doc/app_notes* directory.

Examples

This example includes all the hierarchical instances inside the *spec.abc* hierarchy:

```
config_coverage_scope -include -inst abc.* -spec
```

The following three commands instruct SLEC to analyze coverage only for all spec targets at the top level and for all instances of the modules whose name ends in the suffix *_mult*:

```
config_coverage_scopes -exclude -all
config_coverage_scopes -include -inst spec.*
config_coverage_scopes -include -module -impl *_mult
```

The following commands instruct SLEC to disable coverage on all targets in top-level impl instances beginning with *mod*, except for those that begin with *mod_foo*:

```
config_coverage_scopes -exclude -inst impl.mod*
config_coverage_scopes -include -inst impl.mod_foo*
```

config_find_flop_maps

Enables and configures automatic finding of intermediate maps using simulation signature as hints.

Usage

config_find_flop_maps
[-spec_max_latency *latency*] [-impl_max_latency *latency*] [-exclude_list *list_of_flops_signals*] [-include_list *list_of_flops_signals*] [-signal_list *list_of_signals*] [-outfile *filename*] [-use_mapping_info] [-auto_signals]

Arguments

- -spec_max_latency *latency*

  Specifies the maximum latency of the specification design. This value overrides the default latency computed by SLEC. If not specified, SLEC automatically determines the maximum latency by identifying the maximum latency of the flops/signals in the existing intermediate/output maps.

- -impl_max_latency *latency*

  Specifies the maximum latency of the implementation design. This value overrides the default latency computed by SLEC. If not specified, SLEC automatically determines the maximum latency by identifying the maximum latency of the flops/signals in the existing intermediate/output maps.

- -exclude_list *list_of_flops_signals*

  Specifies the list of flops/signals to be excluded from consideration during the flop/signal map search. Note that this option supports the use of wildcards.

- -include_list *list_of_flops_signals*

  Specifies the list of flops/signals to be considered for mapping by the flop/signal map search. Note that this option supports the use of wildcards.

- -signal_list *list_of_signals*

  By default, the flop/signal map search considers all design flops for mapping. Use this option to specify additional signals to be considered. Note that this option supports the use of wildcards.

- -outfile *filename*

  Specifies the name of the output file. By default, the output of the flop/signal map search is written to: <*workdir*>*/slec_generated_flop_maps.tcl*.

- -use_mapping_info

  If specified, flop/signal map search is instructed to allow SLEC to apply user-provided maps to simplify the normalized machine. While this could lead to a faster flop/signal map search, it could also lead to incorrect search results if any of the maps provided are incorrect.

- -auto_signals

  Instructs SLEC to refine maps by automatically determining a list of signals that are likely to map to either a flop or another signal. SLEC identifies potential maps using its internal simulation.

Description

The config_find_flop_maps command creates a list of candidate intermediate maps which can be used in a subsequent verification run to obtain full proof. Note that this mode disables formal analysis and focuses on finding intermediate maps using simulation signatures as hints.

The config_find_flop_maps command is part of a two-pass flow, where the first pass involves running the config_find_flop_maps command. SLEC then generates a list of candidate intermediate maps within verify. In this pass, bounded-proof/full-proof checking is disabled during verify. In the second pass, the user must source the intermediate maps found by SLEC in the first pass and then run SLEC in the appropriate full_proof/bounded_proof mode to get a proof.

When the config_find_flop_maps command is specified, a candidate set of intermediate maps are identified by the simulation signature. These maps are then refined by deeper simulation and solvers. The final list of intermediate maps are written to the file specified by the -out_file option. If not specified, maps are written to the slec_generated_flop_maps.tcl file. The intermediate maps are listed as equivalence classes. The members of each equivalence class share the same simulation signature. Given that the config_find_flop_maps command does not perform full formal proofs for the maps, all the generated maps cannot be guaranteed to be totally correct.

As mentioned above, a subsequent full_proof verification run can source these intermediate maps to improve the chance of obtaining a full proof. For example:

```
# First run
config_find_flop_maps -spec_max_latency 2 -impl_max_latency 5
verify
...

# Second run
source slec_generated_flop_maps.tcl
verify -mode full_proof
```

config_limited_formal

Limits verification time by constraining ll3 verification time and stopping after low effort engines.

Usage

config_limited_formal [-time_limit *sec*]

Arguments

- -time_limit *sec*

  Specifies the maximum time limit (in seconds) for verifying a single map with the ll3 verification engine. Legal values: 1 or greater. The default value is determined by the global ll3_problem_time_limit.

Description

In some cases the default SLEC flow can lead to long verification run times, making it difficult to quickly check designs for equivalency. To limit verification run time, specify the config_limited_formal command with the desired ll3 time limit for each map. This command also instructs SLEC to stop verification after the low effort engines.

config_property_checks

Enable or disable the synthesis of various monitors for property checking.

Usage

config_property_checks {**-enable** | **-disable**} {**-umr** | **-asc** | **-abr** | **-abw** | **-ise** | **-dbz** | **-cas** | **-all**}

Arguments

- **-enable** | **-disable**

  Enable or disable the specified property check(s). At least one of the following property types must be specified. (The property types correspond to the implicit and explicit property checks described in the check_properties command.)

  - Specify **-umr** to enable/disable property checks for uninitialized memory reads.
  - Specify **-asc** to enable/disable property checks for user assert statements.
  - Specify **-abr** enable/disable property checks for out-of-bounds array and bit-vector reads.
  - Specify **-ise** enable/disable property checks for illegal shift operations.
  - Specify **-dbz** enable/disable property checks for divide-by-zero and modulo-by-zero operations.
  - Specify **-cas** enable/disable property checks for incomplete case statements.
  - Specify **-all** to enable/disable all property checks.

Description

The config_property_checks command enables fine-grained control over which property monitors are synthesized by the front end. The config_property_checks command must be issued before a build_design command. The command affects all the subsequent build_design commands, unless the command is issued again with different options. Using the pair of commands, config_property_checks and check_properties/assume_properties, enables you to control both the synthesis of the property monitors and the solvers in the verification flow.

The config_property_checks command only controls the property monitor synthesis for the C++ front end. Controls for SVA and OVL properties in RTL are not currently supported.

Examples

Enable or disable a particular type of check by specifying the property type:

```
config_property_checks -disable -dbz
```

Specify multiple types in the same command line:

```
config_property_checks -enable -umr -dbz
```

Enable all the checks combined:

```
config_property_checks -enable -all
```

Multiple config_property_checks commands can be specified, with the subsequent commands updating the option of check synthesis for the property types specified with it. The following commands configure the front end to synthesize only ISE (illegal shift) type properties:

```
config_property_checks -disable -all
config_property_checks -enable -ise
```

The config_property_checks command should be specified after the set_verification_mode command and before any build_design command for it to affect the synthesis. For multiple designs, you can configure different commands. For example, the following commands instruct the front end to synthesize array bounds violation checks for the spec design and array bounds violations and uninitialized memory checks for the impl design:

```
config_property_checks -disable -all
config_property_checks -enable -abr -abw
build_design -spec spec.cpp
config_property_checks -enable -umr
build_design -impl impl.cpp
```

config_trace_files

Configures generation of counterexample waveforms and waveforms from simulation-based validation.

Usage

config_trace_files [-directory *dirname*] [-disable] [-dumpmem] [-dont_dumpmem] [-auxsignals] [-synclength] [-format *format*] [-simdump] [-level *level*] [-coverdump] [-storedcxdump]

Arguments

- -directory *dirname*

  Copies all counterexample and waveform trace files to the specified directory in addition to the work directory. If you specify this argument, the view_waveform command reads trace files from this directory instead of the work directory.

- -disable

  Disables waveform generation for all engines. By default, waveform generation is enabled.

- -dumpmem

  Flatttens memory arrays and dumps them into waveforms as separate registers. Memories are not dumped unless they are partially or wholly mapped. Some or all of the memory locations must be included in intermediate maps or memory maps.

  Note:
  Memory arrays that are primary inputs or outputs are always dumped into waveforms.

- -auxsignals

  Adds an auxiliary signal 'trans_start' which counts the number of transactions in the design.

- -synclength

  Forces split designs to trace the same amount of simulated time even if the transaction lengths of the two designs are different.

- -format *format*

  Specifies the format of the waveform trace files. Options are "vcd", "fsdb", and "qwave". Default is "qwave".

- -simdump

  Enables waveform trace dump for simulation-based validation. By default, simulation-based validation does not dump waveform traces unless the validation encounters a falsification.

  Note:
  The global sim_based_validation must be set to one (default) to enable simulation-based validation.

- -level *level*

Restricts dumping of waveforms to signals/ports within the scope of the level defined by <*level*>, where level is calculated from the top level of the scope (applies to both the specification and the implementation designs).

- -coverdump

  Enables waveform trace dump during formal verification. By default, SLEC does not dump formal verification waveform traces unless it encounters a falsification.

- -storedcxdump

  Enables waveform trace dump for stored counterexamples. By default, SLEC does not dump counterexample waveform traces unless it encounters a falsification.

Description

By default, SLEC generates both testbenches and waveform files to demonstrate counterexamples.

Waveform files can be generated during the simulation-based validation phase of verification. Generation of simulation-based validation waveform results are disabled by default, and can be enabled using the -simdump option. To disable generation of waveform files, use the -disable option.

SLEC generates individual files for each design. Note that the same base name cannot be specified for both the specification and implementation waveform files.

Note:
SLEC uses a cycle-based simulation algorithm. It is possible that the value changes might not match the results of an event-driven third-party simulator.

cover_properties

Checks the coverage of the specified properties from the designs.

Usage

cover_properties **-sva** [-spec | -impl] [-list *cover_properties*]

Arguments

- **-sva**

  Checks if the specified user System Verilog SVA cover properties are covered.

- -spec | -impl

  Specifies whether the checks are to be performed for properties in the specification design or implementation design.

- -list *cover_properties*

  Specifies a list of SVA cover properties to monitor. Supports "find" command wildcards.

create_and

Performs a logical AND operation on several single-bit signals or waveforms and returns the resulting waveform handle.

Usage

create_and [-name *inst_name*] ***signal_1 signal_2*** ...

Arguments

- -name *inst_name*

  Output signal name. If you do not provide this argument, the auto-generated name follows the format:

  ```
  INST_and_xtor_inst_<n>_out_ss=<sample_start>
  ```

  where <*n*> is a sequentially assigned ID value and <*sample_start*> identifies the sample start time (typically 0).

- ***signal_n***

  Single-bit signal to input to the AND function. You can enter two or more signals. The specified list of signals must be from either the specification design or the implementation design; however, the list cannot include signals from both designs.

Description

The create_and command performs a logical AND operation on several single-bit signals or waveforms and returns the resulting single-bit waveform handle. Inputs can be signals from either the specification or implementation design, but not both.

Examples

Example 1

The following example creates an AND function of two signals in the specification and maps them to a single signal in the implementation.

```
set sync_clk_check [create_and spec.inst1.read spec.inst2.read]
create_map -intermediate $sync_clk_check impl.read
```

Example 2

The following example generates an AND signal from two sampled waveforms.

```
create_and [create_waveform -sample_start 1 spec.x]
  [create_waveform -sample_start 2 spec.y]
```

create_black_box

Creates a black box around one or all instances of a module.

Usage

create_black_box {**-spec** | **-impl**} { **-inst** ***inst*** | **-module** ***module*** }

Arguments

- **-spec** | **-impl**

  Specifies the name of the library containing the module. Note that this option cannot be specified along with the -inst option.

- **-inst** ***inst*** | **-module** ***module***

  Specifies the name of the instance or module to be blackboxed.

  Use -instance to specify the name of the instance to blackbox starting with the top module instance spec or impl.

  Note:
  The -inst option can only be specified after the design has been built using the build_design command.

  Use -module to specify the name of the module to blackbox. This option requires the -spec or -impl option to be specified.

Description

The create_black_box command removes the internal logic of a module while preserving its interface. Instances of the blackboxed module do not drive their output ports. The input ports of a blackboxed instance become primary output ports of the design, while the output ports of the instance become primary input ports of the design.

The create_black_box command must be used for large memories and for blocks that contain unsynthesizable logic. Such blocks must be blackboxed using the -module option before the corresponding read_design or build_design command are provided.

Note that for SystemC designs, the create_black_box command can only be specified before build_design along with the -module option. For example:

```
create_black_box -spec -module amem
build_design -spec filter.cpp
```

Instances can be blackboxed any time before constraints or maps are specified. SLEC requires that each blackboxed instance in a design is mapped to a correspondingly blackboxed instance in another design. Note that modules/instances with inout ports cannot be blackboxed.

There are two ways to identify the instances to be blackboxed. The first form blackboxes all instances of a module using the -module option. The library containing the module must be specified. For example, the following commands blackbox all instances of the "fast_mem" module found in the specification and implementation libraries.

```
create_black_box -spec -module fast_mem
create_black_box -impl -module fast_mem
```

The -module form of the command can and should be used before the build_design, build_db, read_db and link_db commands.

Because the output ports of the blackboxed module are no longer driven, they should be constrained to appropriate values and set to appropriate clock domains, if necessary.

The second form directly specifies the instance of the module to be blackboxed using the -inst option. Assuming fast_mem_bank is an instance of fast_mem, the following command will blackbox only the fast_mem_bank instance of the fast_mem module.

```
create_black_box -inst spec.mem_unit.fast_mem_bank
```

Usage Notes

- The -inst form of the command can only be used after the database has been built. The -inst option expects the full hierarchical path starting with "spec" or "impl".
- The -inst option does not work with the -spec and -impl options.
- The top instance of the specification or implementation design cannot be blackboxed.

create_black_box_map

Maps a black box instance in one design to a black box instance in another design.

Usage

create_black_box_map ***spec_instance impl_instance***

Arguments

- ***spec_instance impl_instance***

  Hierarchical names of the two instances to be mapped. One instance must be from the specification design and another from the implementation design. Note that the order is not important and wildcard substitution can be used to map more than one pair of instances.

Description

The create_black_box_map command maps an instance in one design to an instance in another design. Ports with the same name are implicitly mapped while mapping instances. Ports which do not map by name can be mapped using the create_map command by specifying the full hierarchical names of the ports.

Assume there is a module "black" in the specification design with ports "in0", "in1", and "out1", which is instantiated in the top module as "spec.black1". In the implementation design, there is a module box with ports "in0", "in1", and "out", which is instantiated in the top module as "impl.box1".

The following commands completely map the two black boxes. The black box map implicitly maps the input ports while the port map explicitly maps the output ports which do not map by name:

```
create_black_box_map spec.black1 impl.box1
create_map -input spec.black1.out1 impl.box1.out
```

For black box instances which match by hierarchical name, an implicit black box map is created. In the previous example, if the implementation instance had been named impl.black1, an implicit black box map would be created. In which case, only one command would have been needed to map all the black box ports.

```
create_map -input spec.black1.out1 impl.black1.out
```

If a black box map is created using an instance already involved in an implicit black box map, the implicit black box map will be removed.

If the maps between the black box ports involve latency, then the latency in mapping must be explicitly specified. Consider the following example.

If the inputs and outputs of the black box in the specification design in cycle 0 map to the corresponding black box ports in the implementation design in cycle 1, the maps must be explicitly specified as follows.

```
create_map -output spec.black1.in0 [create_waveform -sample_start 1 impl.box1.in0]
create_map -output spec.black1.in1 [create_waveform -sample_start 1 impl.box1.in1]
create_map -input spec.black1.out1 [create_waveform -sample_start 1 impl.box1.out]
```

The next example shows how wildcards can be used with the create_black_box_map command.

Assuming a specification design has the instances U1 and U2 blackboxed, and an implementation design also has blackboxed instances named U1a and U2, the following command can be used to map the instances.

```
create_black_box_map spec.U* impl.U%1a
```

Alternatively, these instances could have been mapped using the following two commands.

```
create_black_box_map spec.U1 impl.U1a
create_black_box_map spec.U2 impl.U2a
```

Note:
Information about mapped and unmapped black boxes can be found in the *<workdir>/bbox.log* file.

create_buffer

Creates a buffered signal edge and returns the resulting signal handle.

Usage

create_buffer [-name *inst_name*] ***signal***

Arguments

- ***signal***

  Signal to input to the buffer.

Description

The create_buffer command creates a buffered signal edge and returns the resulting signal handle. The input can be a signal from either the specification or implementation design. You can include the create_buffer command for signal mapping and other SLEC commands requiring a signal edge.

Examples

Consider the following example of the create_buffer command:

```
set module_inst_port_bit [create_buffer spec.inst1.data_in]
create_map -intermediate $module_inst_port_bit impl.data_in
```

The command sequence above confirms that a module input signal in the specification is equivalent to a signal in the implementation that has flattened the corresponding module instance.

create_clock

Creates an ideal clock which can be subsequently attached to clock input ports in a design.

Usage

create_clock **-name *name* -period *period***

Arguments

- **-name *name***

  Specifies the name of the clock being created.

- **-period *period***

  Specifies the period of the clock, in integer units. The minimum clock period is 10 and <*period*> must be an even number.

Description

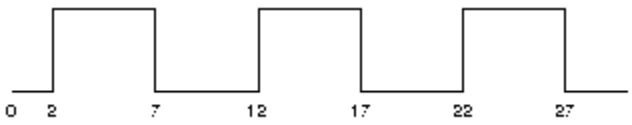
The create_clock command creates a clock waveform, also known as an ideal clock, with the specified clock period and given name. Subsequently, this clock waveform can be associated with input ports or black box output ports which are clocks, using the set_clock_root command.

Both the name and period of the clock must be specified. Clock waveforms always start with value 0. Positive clock edges occur at time 2 and every <period> units thereafter. Negative edges occur <*period*> /2 units after every positive edge.

For example,

```
create_clock -name top_clock -period 10
```

creates an ideal clock named top_clock with the following waveform:



The minimum clock period is 10. The period must be an even number (not an odd number). Only the period of the clock can be specified. The duty cycle of a clock waveform is 50%. By default, there is a default ideal clock named "_Ideal_Clk_" with a period of 100. The _Ideal_Clk_ waveform starts low; it has its first positive edge at 2 and its first negative edge at 52.

The name of the default ideal clock is reserved, and the properties of the default ideal cannot be redefined using the create_clock command.

Examples

This example shows a clock source named MAIN_CLK that is created with a period of 200ns. This clock is then applied to the port CLK_IN on the specification and implementation designs using the set_clock_root command. The set_clock_root command can also be used to modify the polarity.

```
create_clock -name MAIN_CLK -period 200
set_clock_root -clock MAIN_CLK spec.CLK_I
set_clock_root -clock MAIN_CLK impl.CLK_IN
```

Two clocks cannot be created with the same name. If two clock pins are driven by a clock and its inverted waveform, use the "set_clock_root -polarity" option in to distinguish them, as shown in the example below.

```
create_clock -period 100 -name ICLK
set_clock_root -clock ICLK -spec clk_pin
set_clock_root -clock ICLK -polarity 1 -spec inv_clk_pin
```

For more information on associating clocks with clock input ports, refer to the set_clock_root command.

create_concat

Performs an element-wise concatenation operation on several signals or waveforms and returns the resulting waveform handle.

Usage

create_concat [-name *inst_name*] ***signal_1 signal_2*** ...

Arguments

- -name *inst_name*

  Output signal name. If you do not provide this argument, the auto-generated name follows the format:

  ```
  INST_concat_xtor_inst_<n>_out_ss=<sample_start>
  ```

  where <*n*> is a sequentially assigned ID value and <*sample_start*> identifies the sample start time (typically 0).

- ***signal_n***

  Signal to input to the concatenation function. You can enter two or more signals. The specified list of signals must be from either the specification design or the implementation design; however, the list cannot include signals from both designs.

Description

The create_concat command performs an element-wise concatenation operation on several signals or waveforms and returns the resulting waveform handle. Inputs can be signals from either the specification or implementation design, but not both. The bit-width of the resulting waveform is the sum of the bit-widths of the inputs.

Examples

Example 1

The following example concatenates two signals in the specification and maps them to a single signal in the implementation.

```
set sync_clk_check [create_concat spec.inst1.read spec.inst2.read]
create_map -intermediate $sync_clk_check impl.read
```

Example 2

The following example creates an element-wise concatenation of two sampled waveforms.

```
create_concat [create_waveform -sample_start 1 spec.x]
  [create_waveform -sample_start 2 spec.y]
```

create_constant

Creates a constant with a specified value in the specification or implementation design and returns a handle to it.

Usage

create_constant { **-spec** | **-impl** } **-value** ***value***

Arguments

- **-spec** | **-impl**

  Specifies whether the constant is to be created for the specification or the implementation design.

- **-value** ***value***

  Specifies the value of the constant to be created.

Description

The SLEC problem setup only allows mappings between a specification and implementation waveform.

SLEC verifies the equivalence of pairs of signals and ports in the specification and implementation designs and these equivalences are specified using the create_map command. Some verification flows require that SLEC also check that some signals in one or both of the designs are constant values. The create_constant command can be used to create a signal having a specified constant value in either the specification or implementation designs. For example, if SLEC needs to check that the signal "spec.num_entries" is always 10, then the following sequence of commands can be used:

```
set const_tenval [create_constant -impl -value 10]
create_map -output spec.num_entries $const_tenval
```

In this example, the constant was created in the implementation design because a valid output map could only be specified between signals in the specification and implementation designs and a need existed to compare spec.const_tenval with a signal that was always set to 10.

The commonly occurring values 0 and 1 can alternatively be created using the commands create_zero and create_one respectively; these commands do not require the -value option.

Assuming the specification design has output ports named fifo_overflow and tx_done. Further, assume that the output port fifo_overflow should never be set to 1 and that the output port tx_done should always be set to 1 during the specification transaction. In such a situation, the create_constant command can be used to specify the design checks as follows:

```
create_map -output spec.fifo_overflow [create_constant -impl -value 0]
create_map -output spec.tx_done [create_constant -impl -value 1]
```

To use the create_zero and create_one commands instead, specify the following:

```
create_map -output spec.fifo_overflow [create_zero -impl]
create_map -output spec.tx_done [create_one -impl]
```

create_constraint

Creates a constraint on an input port.

Usage

create_constraint [-name *constraint_name*] [-reset] [-num_samples *num_samples*] **-waveform *waveform port***

Arguments

- -name *constraint_name*

  Specifies the name of the constraint. If not specified, a unique name is automatically generated. The name is returned as the result of the command.

- -reset

  Specifies that the constraint is to be applied during reset. When omitted, the constraint is applied during normal operation.

- -num_samples *num_samples*

  Specifies the number of samples to apply per transaction. If not specified, *<num_samples>* defaults to the throughput of the design.

- **-waveform *waveform***

  Specifies the name of the waveform to be used to constrain the input port.

- ***port***

  Specifies the name of the primary input port or black box output port being constrained. This option supports the use of wildcards.

Description

The create_constraint command constrains an input port by associating a waveform with the input port. This command returns the name of the constraint created.

A few points to note about this command:

- If the input is mapped by name to a corresponding port on the other design, then the constraint is applied to the mapped port also.

- Input ports cannot be multiply constrained. The waveforms used to constrain an input port must be explicit or derived from the same design which contains the port.

- The *<port>* specifier refers to a single input port or a sub-field of an input port. If an input port, in, of the specification top module instance is 16-bits wide, then spec.in[0] refers to bit 0 of the port, and creates a constraint which constrains bit 0 of the specification "in" port to a waveform which is always zero.

```
create_waveform -name Always_zero -bitwidth 1 0+
create_constraint spec.in\[0\] -waveform Always_zero
```

- By default, the constraint is applied with transaction scope. With transaction scope, the constraining waveform is restarted at the beginning of each transaction. With global scope, the constraint is applied once only and immediately after reset. Since transaction scope is the default scope, -transaction_scope is unnecessary, but can be used to explicitly document the intended scope.

- When the specification and implementation designs have different throughputs, input constraints should be applied separately to mapped inputs on both designs. For example, a specification design with a throughput of 8 which has an input port "in" can be set to a constant one as follows:

  ```
  create_waveform -name Always_one -bitwidth 1 {1+}
  create_constraint -waveform Always_one spec.in
  ```

  Now, consider another example where the implementation design has a throughput of 16 and an input port called in. The impl.in port is mapped by name to the spec.in port and inherits the constraint from spec.in. However, the spec.in constraint is only effective for 8 cycles because the constraint has transaction scope and the throughput of the specification design is 8. If the intent is to constrain impl.in to one for 16 cycles, then the constraint must be applied explicitly as follows.

  ```
  create_waveform -name Always_one -bitwidth 1 {1+}
  create_constraint -waveform Always_one spec.in
  create_constraint -waveform Always_one impl.in
  ```

- If -reset is specified, the constraint is applied during reset instead of normal operation.

- Ports used only for reset should be constrained to non-reset values for normal operation to keep the design out of reset during equivalence checking.

create_coverage_target

Create a coverage point with the desired polarity.

Usage

create_coverage_target [-polarity 0 | 1] ***signal***

Arguments

- -polarity 0 | 1

  Specifies the desired value that the signal should be proven to be able to take. Legal values are 0 or 1.

- ***signal***

  Specifies the target signal for the command. The signal must have 1-bit width.

Description

The create_coverage_target command instructs the SLEC tool to attempt to prove that the specified signal can be assigned the desired value. If formal engines are reached during verification, then each coverage target will be reported to have one of the following states:

- **Covered** : The coverage point was proven to take the desired polarity value.
- **Not covered up to T** : The coverage point was proven not to take the desired polarity value until transaction T.
- **Uncoverable** : The coverage point was proven not to take the desired polarity value.
- **Not attempted** : Formal engines were interrupted before the coverage point could be considered.

create_invert

Creates an inverted version of a single-bit signal or module port bit and returns the resulting signal handle.

Usage

create_invert [-name *inst_name*] ***signal***

Arguments

- -name *inst_name*

  Output signal name. If you do not provide this argument, the auto-generated name follows the format:

  ```
  invert_<input_signal_name>
  ```

  where <*input_signal_name*> is the name of the input signal.

- ***signal***

  Specifies the name of the signal whose polarity is to be inverted. This must be a one-bit signal or module port bit from either the specification or implementation design.

Description

The create_invert command creates an inverted version of a single-bit signal or module port bit and returns the resulting single-bit signal handle. The input can be a signal from either the specification or implementation design. This command is useful in cases where the polarity of the output port in one design differs from the polarity of the output port in the other design. You can specify this command to generate signals for port maps such as those specified with the "create_map -valid" command.

Examples

In the following example, the specification port out1 is mapped to the inverse of the implementation port out1_inv.

```
create_map spec.out1 [create_invert impl.out1_inv]
```

Consider another example with the "create_map -valid" command. In this example, SLEC verifies the mapping only when the signal impl.OEN is low (when the inverse of impl.OEN is high).

```
create_map spec.out impl.out .valid [create_invert impl.OEN]
```

In addition, SLEC supports the following HDL-specific formats for slices:

```
[create_invert {spec.out1(2 downto 2)}] ## vhdl style slice name
```

```
[create_invert {spec.out1(2 to 2)}] ## vhdl style slice name
```

```
[create_invert {spec.out1[2:2]}] ## verilog style slice name
```

Finally, consider an example where we map an output port bit in the specification to an inverted output port bit in the implementation:

```
set buffered_port_bit [create_buffer spec.inst1.data_out]
set inverted_port_bit [create_invert impl.inst1.data_out]
create_map -intermediate $buffered_port_bit $inverted_port_bit
```

create_map

Maps a waveform, port or signal from the specification design to a waveform, port, or signal in the implementation design.

Usage

create_map ***spec_waveform impl_waveform*** [-name *name*] { **-input** | **-output** | **-intermediate** }
[-num_samples *num_samples*] [-assume_only] [-valid *valid_waveform*] [-induct *induction_depth*]
[-cut]

Arguments

- ***spec_waveform impl_waveform***

  Specifies the names of the waveforms, ports, or signals being mapped between designs. The order is not important. Wildcard substitution may be used to map more than one pair of waveforms, ports, or signals.

- -name *name*

  Specifies the name of the map being created. If not specified, a unique name is generated automatically. Wildcard substitution may be used to generate a list of names corresponding to matching pairs in a wildcarded description. The name or list of names is returned as the result of the command.

- **-input** | **-output** | **-intermediate**

  Specifies the type of map. This could be an input, output, or intermediate map.

  Intermediate maps specify the secondary nature of the map. Intermediate maps are proven only when they are structurally relevant to other output maps.

- -num_samples *num_samples*

  Specifies the number of samples to map per transaction. For input maps, defaults to the minimum throughput value of both designs. For output maps, this defaults to 1.

  Note:
  The -num_samples option cannot be used with the -induct option or with intermediate maps.

- -assume_only

  Indicates that SLEC does not need to verify the validity of specified output map or intermediate map.

  Note that SLEC does not validate the accuracy of the assumption. SLEC blindly follows the assumption set by the user. Marking an invalid map with -assume_only can lead to a false positive or false negative.

- -valid *valid_waveform*

  Indicates that an output check should be performed at a clock cycle only if the *valid_waveform* signal also evaluates to true in the same clock cycle. If the *valid_waveform* signal is false at the clock cycle where the output check is to be performed, then the check passes trivially. In case the waveform is based off a multi-bit signal, the reduction-OR of the bits is checked. See further discussion on ramifications of using -sample_start to create the *valid_waveform* signal.

Note:
The -valid option can be specified multiple times for output and intermediate maps. This map is checked only when all of the valids evaluate to true.

- -induct *induction_depth*

  Specifies the induction depth. Note that this option works only with output and intermediate maps.

- -cut

  Specifies that the intermediate map needs to be cut-at during formal analysis.

Note:
The following restrictions apply to this option:

- This option is valid only for intermediate maps.
- This option is honored only when the mapping is specified between a pair of specification and implementation flops.
- This option cannot be used with the -induct option.

Description

The create_map command is used to map inputs, outputs, or signals between designs. This command returns the name of the map.

Input Maps

Input maps are applied with "transaction scope" which means that the mappings are applied for each transaction. Consider an example where the throughput of the specification design is 3 and the throughput of the implementation design is 4.

If the following command was applied:

```
create_map -input spec.A [create_waveform -sample_start 1 impl.B]
```

then, -num_samples will be 3 because it is the smaller of the throughputs. If the waveforms of spec.A and impl.B are as follows:

```
spec.A A0 A1 A2 A3 A4 A5 A6 A7 A8 ...
impl.B B0 B1 B2 B3 B4 B5 B6 B7 B8 ...
```

then, the maps will be applied as follows:

```
Transaction-1: A0 == B1, A1 == B2, A2 = B3 (that is, B0 is not mapped)
Transaction-2: A3 == B5, A4 == B6, A2 = B7 (that is, B4 is not mapped)
```

Note:
The numeric suffixes indicate the cycle number.

By default, input ports with the same name in both designs are implicitly mapped. Implicit maps are applied from the beginning cycle of a transaction with a default number of samples equal to the minimum throughput of the two designs. An implicit map can be removed using the unmap command.

Note that there is no other way to remove or delete a map that has been explicitly added.

More complex relationships between ports are specified using waveforms.

Output Maps

Mapped output waveforms are compared with transaction scope. The mapped waveforms are restarted at the beginning of each transaction.

By default, output ports with the same name in both designs are implicitly mapped. Implicit maps are compared output latency cycles after the beginning of a transaction, where the output latency is specified by the globals spec_output_latency and impl_output_latency for specification and implementation outputs respectively.

Sequentially different behavior can be captured by using appropriately created waveforms, for example:

```
create_waveform -name Spec_Sample -sample_freq 2 -sample_start 0 -sample_end 5 spec.data_out
create_waveform -name Impl_Sample -sample_freq 3 -sample_start 0 -sample_end 8 impl.data_out
create_map -output -num_samples 3 Spec_Sample Impl_Sample
```

This example compares two sampled outputs. The outputs are checked three times per transaction. The specification's data_out port is compared in cycles 0, 2, and 4 with the implementation's data_out port in cycles 0, 3, 6. The implicit map by name for the data_out ports is replaced by this map.

Using the -valid Option

The -valid option is used to produce conditional checks on the output map. For the previous example, replacing the map as follows will compare the sampled waveforms only in output check cycles when the impl.valid waveform is also high.

```
create_map -output -num_samples 3 -valid impl.valid Spec_Sample Impl_Sample
```

Note that multiple valid maps can also be specified. Consider the following example:

```
create_map -output -num_samples 3 -valid impl.valid -valid impl.valid1 \
-valid impl.valid2 Spec_Sample Impl_Sample
```

In this example, SLEC checks the output map only when all the valid conditions evaluate to true.

How is valid_waveform Sampled?

To determine whether the outputs must be compared, only the value of the valid_waveform signal in the output check cycles is sampled, and its value at other cycles are ignored. Consider another create_map command in the context of our example:

```
create_map -output -num_samples 3 -valid spec.valid -valid impl.valid
 Spec_Sample Impl_Sample
```

Let us write sig@k to denote the value of signal sig at cycle k. This means that:

- *spec.data_out@0* is compared with *impl.data_out@0* but only if *spec.valid@0* and *impl.valid@0* are both high; otherwise the comparison passes trivially.

- spec.data_out@2 is compared with impl.data_out@3 but only if spec.valid@2 and impl.valid@3 are both high; otherwise the comparison passes trivially.

- spec.data_out@4 is compared with impl.data_out@6 but only if spec.valid@4 and impl.valid@6 are both high; otherwise the comparison passes trivially.

Note that spec.valid is sampled exactly in the cycles where spec.data_out is also sampled. Likewise, impl.valid is sampled exactly in the cycles where impl.data_out is also sampled. As in other cases, the sampling cycles for valid signals may be shifted via the create_waveform command with the -sample_start option. Consider the following commands:

```
create_waveform -name Spec_Valid -sample_start 5 spec.valid
create_waveform -name Impl_Valid -sample_start 7 impl.valid
create_map -output -num_samples 3 -valid Spec_Valid -valid Impl_Valid \
Spec_Sample Impl_Sample
```

Let us again write sig@k to denote the value of signal sig at cycle k. This means that:

- spec.data_out@0 is compared with impl.data_out@0 but only if spec.valid@5 and impl.valid@7 are both high; otherwise the comparison passes trivially.

- spec.data_out@2 is compared with impl.data_out@3 but only if spec.valid@7 and impl.valid@10 are both high; otherwise the comparison passes trivially.

- spec.data_out@4 is compared with impl.data_out@6 but only if spec.valid@9 and impl.valid@13 are both high; otherwise the comparison passes trivially.

Using the -induct Option

SLEC allows the user to model a scenario where a map is assumed to be true for one transaction and is guaranteed to be equal in the next transaction. This is called 1-step induction and can be specified as "-induct 1". Similarly, *K*-step induction entails assumption maps for the first *K* transactions, beginning at the latency provided and a guarantee map for the *K+1th* transaction. *K* is said to be the induction depth of the map.

Consider the following example where the throughput for the specification and implementation design is 1.

```
create_map -output -induct 2 spec.f impl.f
```

This translates into the following assume only maps:

```
create_map -output -assume_only spec.f impl.f
create_map -output -assume_only [create_waveform -sample_start 1 spec.f] \
 [create_waveform -sample_start 1 impl.f]
```

and, the following guarantee map:

```
create_map -output [create_waveform -sample_start 2 spec.f] \
 [create_waveform -sample_start 2 impl.f]
```

Note:
If induct maps are used to map flops between designs, the symbolic reset values of the flops have to be explicitly mapped using the create_symbolic_reset_group command.

Intermediate Maps

Output maps are primary proof objectives for SLEC. SLEC is bound to verify the output maps. Intermediate maps allow the user to specify secondary proof objectives, that is, intermediate maps are proven only when they are in the fan-in of other primary proof objectives. Intermediate maps can be used to specify the mapping between signals in the specification and implementation designs. If intermediate maps are used to map flops with symbolic reset values, then the create_symbolic_reset_group command should be used to map the reset values.

The following example shows how flops can be mapped using intermediate maps.

```
create_map -intermediate spec.f impl.f
create_symbolic_reset_group -if_symbolic {spec.f impl.f}
```

To implicitly map, all the flops in the design, use the create_namemap_rule command.

Type Mismatch

When inputs, outputs, or signals of designs are compared, there may be type differences on the ports being mapped. For example an sc_int in a SystemC design may be mapped to a reg[31:0] in a Verilog design. Type mismatches may be mismatches in sign, width, or both.

SLEC checks values based on bit equivalence.

Consider two ports being mapped. The wider port has a width of m bits while the narrower port has a width of n bits.

For input maps, the n bits of the narrower type are mapped to the corresponding lower n bits of the wider type. The additional upper m-n bits of the wider type are not constrained. If the sign of the types are different, a warning is issued, but the lower n bits of the two ports are still bit-wise mapped.

For output and intermediate maps, the n bits of the narrower type are mapped and compared with the corresponding lower n bits of the wider type. The upper m-n bits of the wider type are compared with all 0s. If the sign of the types are different, a warning is issued but the comparisons are still made.

The example below illustrates how SLEC performs the checks for output maps.

```
output signed [3:0] o1_sig;
output [3:0] o2_sig;
output signed [4:0] o3_sig;
output [4:0] o4_sig;
```

Specifically with the output declarations as above, the following values are deemed to be equal (because all of the bits match with zero-extension):

```
o1_sig = 1111 (-1)  vs  o2_sig = 1111 (15)
o1_sig = 1111 (-1)  vs  o3_sig = 01111 (15)
o2_sig = 0111 (7)   vs  o4_sig = 00111 (7)
```

On the other hand, the following pair of values are not deemed to be equal:

```
o1_sig = 1111 (-1) vs o3_sig = 11111 (-1)
```

Using Wildcards

The create_map command support wildcard substitution and matching. For example, if the instance hierarchies in the specification and implementation design match up completely and one wants to map all of the outputs within spec.i1 to the similarly named outputs within impl.i1, one can specify:

```
create_map -flop spec.i1.**.* impl.i1.%1.%2
```

A few points to note about the above syntax:

- The pattern ** in <spec.i1.**.*> can match 0 or more hierarchies.
- The pattern * in <spec.i1.**.*> will match any string which is a flop. Pattern matching will be applied only to outputs because the create_map -output command is used.

  For example, in the above context, the pattern <spec.i1.**.*> will match the following flops: <spec.i1.p1, spec.i1.i2.i3.i4.i5.p2, spec.i1.i1.p3> but will not match <spec.i2.p4> and <spec.p5>.
- For each string that is matched by spec.i1.**.*, the string matched by **, that is, the first wildcard is stored in %1 and the string matched by *, that is, the second wildcard, is stored in %2. For example, if the list of all of the specification design flops matching spec.i1.**.* was: <spec.i1.p1, spec.i1.i2.i3.i4.i5.p2> and <spec.i1.i1.p3>, then SLEC will look for corresponding implementation design flops, that is, <impl.i1.p1, impl.i1.i2.i3.i4.i5.p2> and <impl.i1.i1.p3>.
- It will also create an output map between the pair of outputs when the corresponding flop exists in the implementation design. For instance, if the flop <impl.i1.i1.p3> does not exist but the first two exist, then the following output maps will be created:

  ```
  create_map -output spec.i1.p1 impl.i1.p1
  create_map -output spec.i1.i2.i3.i4.i5.p2 impl.i1.i2.i3.i4.i5.p2
  ```

Note:
Observe the following details about using wildcards for this command:

- SLEC writes out details of exact matches and substitutions in the *slec.cmd* file.
- Although "create_map -output" can be used to map intermediate signals in the design, when used with wildcards, it only maps output ports.
- Wildcards cannot be used with the "create_map -intermediate "command.

create_memory_map

Perform optimized modeling of the specified arrays in the specification and implementation designs and map them during verification. This command must be specified before the build_design command.

Usage

create_memory_map [-symbolic | -hash | -sparse] [*symbolic_memory_options* | *hash_memory_options* | *sparse_memory_options*] ***specification_design_array implementation_design_array***

Arguments

- -symbolic | -hash | -sparse

  Indicates the type of memory model that will be used to model the specified arrays.

  By default, all arrays are modeled using the Symbolic Memory Model. Use the -symbolic option when the memories are expected to be equivalent for each transaction and cutting memories at each transaction is not expected to cause a false negative, that is, a pessimistic falsification.

  Use the -hash option to model the array as a Hash Memory Model. While this model might not always provide a complete verification, it works well if the use of the address generation logic is deemed to be agnostic to a particular address or set of addresses. For example, if the specified hash_table_size is 64 and the size of the original array is 256, then read and write operations beyond address 63 are aliased to a location between 0 and 63 (inclusive of address locations 0 and 63).

  The -sparse option is no longer supported. This option was used to model the array as a Sparse Memory Model.

- ***specification_design_array implementation_design_array***

  Hierarchical names of the two arrays which need to have the specialized/efficient modeling. The first array needs to be from the specification design and the second array from the implementation design.

Symbolic Memory Options

- -spec_write_memory_latency *spec_write_latency*

  Specifies the latency of the write to the specification memory within a transaction.

- -impl_write_memory_latency *impl_write_latency*

  Specifies the latency of the write to the implementation memory within a transaction.

- -induct *induction_depth*

  Specifies the induction depth of the map. Induction implies a map similar to an assume-guarantee. When specified, SLEC uses induction-based solvers to verify the memory map.

Hash Memory Options

- -hash_table_size *size* | -size *size*

  Specifies the size of the hash-memory model used to model the specified arrays, if the -hash option has been specified. This argument is optional and default size is 32.

- -spec_write_memory_latency *spec_write_latency*

Specifies the latency of the write to the specification memory within a transaction.

- -impl_write_memory_latency *impl_write_latency*

  Specifies the latency of the write to the implementation memory within a transaction.

Sparse Memory Options (OBSOLETE)

- -num_reads *reads_per_cycle*

  Specifies the number of memory read operations.

- -num_writes *writes_per_cycle*

  Specifies the number of memory write operations.

- -spec_write_memory_latency *spec_write_latency*

  Specifies the latency of the write to the specification memory within a transaction.

- -impl_write_memory_latency *impl_write_latency*

  Specifies the latency of the write to the implementation memory within a transaction.

- -spec_read_memory_latency *spec_read_latency*

  Specifies the latency of the read to the specification memory within a transaction.

- -impl_read_memory_latency *impl_read_latency*

  Specifies the latency of the read to the implementation memory within a transaction.

Description

True modeling of the large arrays in the designs being equivalence-checked can adversely impact the performance of formal-verification. The create_memory_map command tells SLEC to perform specialized modeling and mapping of the specified arrays. The use of the specialized modeling can provide a significant boost to the performance of build_design and the solvers within verify.

Symbolic Memory Modeling

Consider the following example.

```
create_memory_map -symbolic spec.v_array impl.v_array.M
build_design -spec ...
build_design -impl ...
```

This example shows how the arrays spec.v_array and impl.v_array.M are modeled as symbolic memories and subsequently mapped during verification. Consider another example. This example shows how to specify the write latency associated with the implementation design memory.

```
create_memory_map -symbolic spec.v_array impl.v_array.M \
 -impl_write_memory_latency 2
```

In this example, SLEC verifies the equivalence of the arrays spec.v_array in cycle-0 of any transaction with impl.v_array.M in cycle-2 of the corresponding implementation transaction.

In case verification involves verifying a specification design against a pipelined implementation, use the -induct option. For example, if the depth of the pipeline is 3, use the following command:

```
create_memory_map -symbolic spec.coeff_arr impl.coeff_arr.M \
 -impl_write_memory_latency 2 -induct 3
```

Increasing the induction depth, reduces the possibility of getting false negatives attributed to cutting of the memories at transaction boundaries.

build_design Output

On successfully processing an array in build_design and modeling it as a symbolic memory, SLEC displays a message confirming the same.

```
[CPT-MMA] Symbol .v_array' has been associated with 'symbolic' memory
 model. The symbol is declared at (/users/jbond/007/hdtv.cpp:1024)
```

Reset Values for Symbolic memories

Symbolic memories can only be reset to X's. For arrays modeled using symbolic memory, the *<workdir>/reset.log* file, generated by SLEC, will have an entry as follows:

```
# set_reset_value -list spec.M -value 'hX@* ; # Symbolically-mapped
```

Note:
The @* implies that the preceding 'hX value applies to all locations of the symbolic memory.

Falsifications Involving Maps on Symbolic Memories

SLEC shows the list of *<address, value>* pairs for each of the memories involved in falsifying a symbolic memory map. For example:

```
[FALSIFIED 3] Output-pair:spec.v_array(1, 1) ('hX@mapped 'h00@0 'h02@7) @
 353 impl.M.ram.M (1, 1) ('hX@mapped 'h00@0 'h01@7) @ 353
```

The above should be interpreted as: the counterexample found by SLEC drives the locations 0 and 7 of spec.v_array to 'h00 and 'h02 respectively; whereas the corresponding locations in the mapped impl.M.ram.M are 'h00 and 'h01 respectively. Note that the 'hX@mapped implies that, unless explicitly specified, the mapped locations of the specification and implementation memories have mapped symbolic values.

Waveforms

By default, SLEC does not dump waveforms for memories. To instruct SLEC to write out the waveforms of the memories, the config_trace_files -dumpmem option needs to be specified. SLEC cuts at symbolic memories for every transaction; so the values of memory locations may change from one transaction to another, even if the design is not explicitly performing a write. SLEC introduces an artificial X band to denote that the following value is a symbolic cut value.

Testbenches

SLEC-generated testbenches cannot simulate the effect of symbolic memories because the cuts on memories cannot be replicated. As a consequence, the testbenches may not be able to reproduce a SLEC falsification when it depends on the specific cut values.

Coding Guidelines

The complexity of the symbolic memory model is proportional to the number of read and write operations performed in one transaction. Consequently, the recommendations for using symbolic memories are:

1. Symbolic memory modeling works best when the number of read/write operations in a transaction are far less than the size of the array. Because of this, symbolic memories do not work well with max-throughput verification.

2. Code which iterates over all of the locations of a memory should be avoided. For example, code to initialize all of the locations of a memory to a certain value. In symbolic memory modeling, the same locations of the mapped arrays or memories will be initialized to the same symbolic values by SLEC.

Restrictions

SLEC applies the following restrictions to symbolic memories:

1. Multi-way mappings involving symbolic memories are not supported.

2. Symbolic memories cannot be used in combination with the hash or sparse memory modeling style in the same verification script.

3. The sizes of the arrays or memories being modeled as symbolic memories can be different. However the data widths of the arrays being modeled as symbolic memories have to be the same.

4. Complex memory layouts are not supported. This occasionally happens when HLS tools merge data or split data along data or address axes.

Hash Memory Modeling

Consider an array valarray in the function fft specification design which has been declared static. The instance-name of the array will be spec.fft@valarray. If the RTL has a memory corresponding to this array, called mem with the instance-name impl.DUT.fft_proc.mem, then the create_memory_map will be used as follows:

```
create_memory_map -hash spec.fft@valarray impl.DUT.fft_proc.mem
```

If the memory in the implementation design get values corresponding to spec.fft@valarray with a latency of 2 and a -cut needs to be additionally specified, the create_memory_map will be invoked as follows:

```
create_memory_map -hash -cut -impl_write_memory_latency 2 \
 spec.fft@valarray impl.DUT.fft_proc.mem
```

Use the global infer_memory_map_latency to specify that SLEC should infer the appropriate write-latencies for the specification and implementation arrays. For more information on this global, refer to the documentation for infer_memory_map_latency.

Refer to the section on the global select_address_slice_number for information on ensuring the desired verification coverage is obtained with hash memory modeling.

Sparse Memory Modeling

SLEC no longer supports the use of the -sparse option with the create_memory_map command. Calypto suggests using the -symbolic option instead to model the memory as a symbolic memory.

create_namemap_rule

Specifies a rule to be used during name-based flop mapping in SLEC.

Usage

create_namemap_rule { **-default** | **-specrule** ***spec_rule*** **-implrule** ***impl_rule*** | **-unmap_all** } [-exclude]
[-rulename *rulename*] [-portmap]

Arguments

- **-default**

  Creates the default name-mapping rule in SLEC. The default rule is [-specrule %s -implrule %s]. If this option is specified, the "-specrule <*spec_rule*> -implrule <*impl_rule*>" option cannot be provided.

- **-specrule *spec_rule* -implrule *impl_rule***

  The <*spec_rule*> and <*impl_rule*> strings can consist of legal Verilog characters and a special character sequence %s. %s will match the entire/portion of the non-hierarchical name of a flop.

  For the <*spec_rule*> string, there can be at most one occurrence of %s while the impl_rule string allows the %s character to appear anywhere without a limit on the number of times it can be used.

  These strings can also contain references to hierarchical names so the string can contain a hierarchy divider (.). If referring to a hierarchy, omit the specification or implementation hierarchy in the name, as the case may be. No wildcarding or regular expression is allowed on the <*spec_rule*> string.

- **-unmap_all**

  If specified, indicates that SLEC should not perform any name-based mapping for ports as well as flops. Once the "create_namemap_rule -unmap_all" command has been called, no further name-based mapping of ports/flops will be processed. All subsequent calls to the create_namemap_rule will be ignored.

- -exclude

  During the name-based mapping, flop-pairs that match the <*spec_rule*> and <*impl_rule*> strings will not be mapped. This will cause the flop-pair, which would otherwise have been mapped by another create_namemap_rule command, to be not mapped. This is optional.

- -rulename *rulename*

  Associates the name <*rulename*> with the rule. It is used for reporting purposes.

- -portmap

  If specified, indicates that the rule specified with this command is to be applied on ports.

Note:
By default, all rules specified using the create_namemap_rule command apply to flop maps.

Description

The create_namemap_rule command creates a rule to be applied during the name-mapping step in verify. It returns the list of rule names created to the Tcl prompt.

The <*spec_rule*> string specifies a string that should match flops in the specification design. If the rule is non-hierarchical, then SLEC tries to match the rule to flops in every hierarchy of the specification design. The implementation flop name is created by applying the <*impl_rule*> to the spec_name (where %s matches the flop name). The implementation flop-rulename is searched in the implementation design to get a match and perform the mapping/unmapping step.

If multiple rules match a specification flop, then the first rule that matches gets applied. A pair of flops which could be mapped by the rules specified through create_namemap_rule would not be mapped if any one of them has been explicitly mapped through create_map command.

This command can be used to map flops in the specification design to that in the implementation design, for example:

```
slec> create_namemap_rule -rulename scalar_rule -specrule %s -implrule %s_reg
```

scalar_rule will map up flops in the specification design like spec.a, spec.b.c, spec.d.e.f to impl.a_reg, impl.b.c_reg, impl.d.e.f_reg respectively. The list {scalar_rule} is returned as the command result.

An example of using the -portmap option is as follows:

```
slec> create_namemap_rule -portmap -specrule %s -implrule %s_impl
```

This rule maps the ports spec.in1, spec.in2 to the impl.in1_impl and impl.in2_impl ports respectively.

create_nand

Performs a logical NAND (not-AND) operation on several single-bit signals or waveforms and returns the resulting waveform handle.

Usage

create_nand [-name *inst_name*] ***signal_1 signal_2*** ...

Arguments

- -name *inst_name*

  Output signal name. If you do not provide this argument, the auto-generated name follows the format:

  ```
  INST_and_xtor_inst_<n>_out_ss=<sample_start>
  ```

  where <*n*> is a sequentially assigned ID value and <*sample_start*> identifies the sample start time (typically 0).

- ***signal_n***

  Single-bit signal to input to the NAND function. You can enter two or more signals. The specified list of signals must be from either the specification design or the implementation design; however, the list cannot include signals from both designs.

Description

The create_nand command performs a logical NAND operation on several single-bit signals or waveforms and returns the resulting single-bit waveform handle. Inputs can be signals from either the specification or implementation design, but not both.

Examples

Example 1

The following example creates a NAND function of two signals in the specification and maps them to a single signal in the implementation.

```
set sync_clk_check [create_nand spec.inst1.read spec.inst2.read]
create_map -intermediate $sync_clk_check impl.read
```

Example 2

The following example generates a NAND signal from two sampled waveforms.

```
create_nand [create_waveform -sample_start 1 spec.x]
  [create_waveform -sample_start 2 spec.y]
```

create_nor

Performs a logical NOR (not-OR) operation on several single-bit signals or waveforms and returns the resulting waveform handle.

Usage

create_nor [-name *inst_name*] ***signal_1 signal_2*** ...

Arguments

- -name *inst_name*

  Output signal name. If you do not provide this argument, the auto-generated name follows the format:

  ```
  INST_or_xtor_inst_<n>_out_ss=<sample_start>
  ```

  where <*n*> is a sequentially assigned ID value and <*sample_start*> identifies the sample start time (typically 0).

- ***signal_n***

  Single-bit signal to input to the NOR function. You can enter two or more signals. The specified list of signals must be from either the specification design or the implementation design; however, the list cannot include signals from both designs.

Description

The create_nor command performs a logical NOR operation on several single-bit signals or waveforms and returns the resulting single-bit waveform handle. Inputs can be signals from either the specification or implementation design, but not both.

Examples

Example 1

The following example creates a NOR function of two signals in the specification and maps them to a single signal in the implementation.

```
set sync_clk_check [create_nor spec.inst1.read spec.inst2.read]
create_map -intermediate $sync_clk_check impl.read
```

Example 2

The following example generates a NOR signal from two sampled waveforms.

```
create_nor [create_waveform -sample_start 1 spec.x]
  [create_waveform -sample_start 2 spec.y]
```

create_one

Creates a constant with a value of 1.

Usage

create_one [-spec | -impl]

Arguments

- -spec | -impl

  Specifies whether the constant created is to be mapped to an output port in the specification design or the implementation design.

  Use the -spec option to map the constant to an output port in the implementation design. To map the constant to an ouput port in the specification design, use the -impl option.

Description

Specifies whether the constant created is to be mapped to an output port in the specification design or the implementation design.

Use the -spec option to map the constant to an output port in the implementation design. To map the constant to an ouput port in the specification design, use the -impl option.

The command is typically used with the create_map command to map output ports against a constant value. This example maps spec.valid_out to the value 1.

```
create_map -output spec.valid_out [create_one -impl]
```

Port spec.valid_out will be compared to 1 after LT cycles, where LT is the number of clock cycles specified by spec_output_latency and will then be compared every TP cycles, where TP is the value specified by spec_throughput.

create_or

Performs a logical OR operation on several single-bit signals or waveforms and returns the resulting waveform handle.

Usage

create_or [-name *inst_name*] ***signal_1 signal_2*** ...

Arguments

- -name *inst_name*

  Output signal name. If you do not provide this argument, the auto-generated name follows the format:

  ```
  INST_or_xtor_inst_<n>_out_ss=<sample_start>
  ```

  where <*n*> is a sequentially assigned ID value and <*sample_start*> identifies the sample start time (typically 0).

- ***signal_n***

  Single-bit signal to input to the OR function. You can enter two or more signals. The specified list of signals must be from either the specification design or the implementation design; however, the list cannot include signals from both designs.

Description

The create_or command performs a logical OR operation on several single-bit signals or waveforms and returns the resulting waveform handle. Inputs can be signals from either the specification or implementation design, but not both.

Examples

Example 1

Consider the following example of the create_or command:

```
set active_signal_check [create_or spec.inst1.active1 spec.inst2.active2]
create_map -intermediate impl.inst.active $active_signal_check
```

The command sequence above confirms that when a signal in the implementation is active (high), one of two signals in the specification is also active.

Example 2

The following example creates a logical OR function between a signal in the implentation design, and a sampled portion of another.

```
create_or impl.u1.E_48 [create_waveform -sample_start 3 impl.u1.u0.z]
```

create_output_map

Maps an output port from the specification design to an output port in the implementation design. Additionally, provides an option to specify latency and inductive values.

Usage

create_output_map **-spec_port *spec_port*** **-impl_port *impl_port*** [-spec_latency *spec_latency*] [-impl_latency *impl_latency*] [-induct *induct_depth*]

Arguments

- **-spec_port *spec_port***

  Specifies the name of the specification design output port to be mapped.

- **-impl_port *impl_port***

  Specifies the name of the implementation design output port to be mapped.

- -spec_latency *spec_latency*

  Specifies the start latency for the specification port.

- -impl_latency *impl_latency*

  Specifies the start latency for the implementation port.

- -induct *induct_depth*

  Generalizes the ports maps to the specified sequential depth.

Description

The create_output_map command can be used in place of the create_map -output command when latency and inductive values also need to be provided during verification. For example:

```
create_output_map -spec_port spec.y -impl_port impl.y -impl_latency 2
```

In this example, the create_output_map command, maps the output port y in the specification design to a 2-cycle delayed version of the output port y in the implementation design.

This command is equivalent to issuing the following commands:

```
set impl_waveform [create_waveform -sample_start 2 impl.y]
create_map -output spec.y $impl_waveform
```

create_state_encoding

Used to specify a map involving a combinational function between one or more flops in the specification design and one or more flops in the implementation design. Returns the name of the state encoding.

Usage

create_state_encoding **-name** ***name*** [-cut] **-module** ***module port_connection_list***

Arguments

- **-name** ***name***

  User-specified name for this encoding. SLEC will use this name in future references to this state-encoding in log-files.

- -cut

  Specifies that SLEC should cut at this state-encoding while performing formal analysis.

- **-module** ***module***

  Name of an encoding module from the cons library. The module must be combinational.

- ***port_connection_list***

  The port connection list for the module instance, either as an ordered list of flops/waveforms sampling flops with a latency or a list of Verilog-style named port connections with the actuals being flops or waveforms sampling flops with fixed latencies.

Description

The create_state_encoding command establishes a combinational relationship between state in one design and state in the other design. For example, a binary-encoded state in one design may be equivalent to a one-hot encoded state in the other design. The command may also be used to establish a mapping between flops that hold signed numbers but are of different sizes.

Sometimes flops do not map directly across designs but have different encodings for the same state. For example, consider a case where the specification design has a two bit flop, spec.a, which needs to be mapped to a 4-bit flop, impl.b, in the implementation design, where impl.b represents the one-hot encoding of spec.a. The following combinational Verilog module represents the encoding relationship:

```
module one_hot_encoder_4(in, out);
 input[1:0] in
 output[3:0] out;

 assign out = { in[1] & in[0],
      in[1] & !in[0],
      !in[1] & in[0],
      !in[1] & !in[0]
      }
endmodule
```

The following conditions must be met:

- The encoding module must be combinational.
- There must be at least one input and one output in the module, and all inputs must be associated with flops/waveforms sampling flops with the same latency in one design, while all outputs must be associated with flops/waveforms sampling flops with the same latency in the other design.
- In case the latencies associated with both the specification and implementation flops are both zero, then the flops specified in the port list must be either all reset or all not reset in both designs. If all reset, the reset values should conform to the state encoding.

  For example, in the one-hot state encoding example mentioned above, if the reset value of spec.a is 2'b11, then the reset value of impl.b should be 4'b1000.
- The types and sizes of the flops/sampled flop-waveforms must exactly match the types and sizes of the corresponding module ports. For example, a Verilog integer register or a SystemC sc_int object are both signed integer type flops.

Examples

Example 1: The simplest state encoding is one between sign-extended values which may be stored in different size registers. In this example, the module ext_9_to_16 is used to map spec.regA to impl.regB. The module is first read into the -cons library and then mapped using the create_state_encoding command.

```
build_design -cons ext_9_to_16.v
create_state_encoding -module ext_9_to_16
 { .in(impl.regB) .out(spec.regA) }
```

The verilog code for module ext_9_to_16 represents, in combinational terms, the relationship between the input and output ports; in this case it is simply a sign extension.

```
module ext_9_to_16(in, out);
   input signed [8:0] in;
   output signed [15:0] out;

   assign out = in;
endmodule
```

Example 2: A more complex relationship would be that mapping a 3-bit one-hot FSM in the specification design to a binary encoded version of the same state in the implementation design.

```
create_state_encoding -module hot_to_bin {  .in2(spec.flopR8) \
                                            .in1(spec.flopR4) \
                                            .in0(spec.flopR2) \
                                            .out(impl.state) }
```

The code for the module hot_to_bin should have been previously read in to the constraints library as defined by the -cons option. An example of this code is shown below in Verilog.

```
module hot_to_bin(in2, in1, in0, out);
input  in2. in1, in0;
output [1:0] out;
```

```
always@(in2 or in1 or in0)
begin
   case ({in2, in1, in0})
      3'b000: out=2'b0;
      3'b001: out=2'b1;
      3'b010: out=2'b2;
      3'b100: out=2'b4;
      default: out=2'bX;
   endcase
end
endmodule
```

Example 3: This example shows how state encoding is established for a flop in the specification design spec.f1 with a latency of 1 and a flop impl.f2 in the implementation design with a latency of 2, where the latter is a sign extension of the former.

```
module sign_extend(in, out);
  input  signed [7:0] in;
  output signed [31:0] out;

  assign out = in;
endmodule
...
build_design -cons sign_extend.v
create_waveform -name spec_f1_1 -sample_start 1 spec.f1
create_waveform -name impl_f2_2 -sample_start 2 impl.f2
create_state_encoding -name sign_extend1 -module sign_extend \
       { .in(spec_f1_1)  .out(impl_f2_2) }
```

To improve the performance of solvers within SLEC, the user can specify that SLEC cut at the flops involved in a state encoding and as a result reduce the sequential complexity. Cutting at a flop (involved in the state encoding) means that SLEC drives in a fresh symbolic value at the associated Q pin at the cycle in which the flop is compared.

create_symbolic_reset_group

Applies the same symbolic value to the list of flops, latches, and memories.

Usage

create_symbolic_reset_group ***list_of_flops*** [-force_x]

Arguments

- ***list_of_flops***

  A TCL list of flops, latches, and memory locations specified by their hierarchical instance names.

- -force_x

  Forces the reset values of the specified flops to X's if the reset values are already concrete values.

Description

The create_symbolic_reset_group command can be used to assign the same symbolic reset value to flops, latches, and memory locations. The following command assigns the same symbolic reset value to the flops in the list.

```
create_symbolic_reset_group {spec.f1 spec.f2 impl.f1 impl.f3}
```

The following restrictions apply to flops used in a create_symbolic_reset_group command:

- No flop should be involved in a state encoder.
- No flop should be involved in a map where the other flop has a zero-latency but does not have the same symbolic-reset.

```
create_symbolic_reset_group {spec.f1 impl.f2}

# generates an error because the flop impl.f1 does not have the
# same symbolic reset value as spec.f1

create_map -intermediate spec.f1 impl.f1
```

create_symbolic_value

Creates a symbol of the specified bit-width and sign, which can be used to subsequently constrain one or more ports using the set_constant command.

Usage

create_symbolic_value **-name *name* -bitwidth *bitwidth*** [-resetval *resetval*] [-signed]

Arguments

- **-name *name***

  Specifies the name of the symbol being created.

- **-bitwidth *bitwidth***

  Specifies the bit-width of the symbol to be created.

- -resetval *resetval*

  Specifies the value to use during the reset-sequence to constrain the input-ports on which the created symbolic-value is applied using set_constant.

- -signed

  Specifies that the symbol to be created is of signed type.

Description

The create_symbolic_value command creates a symbol which can be used in subsequent set_constant commands to constrain one or more input-ports to a fixed symbolic-value. The constrained ports will retain the same symbolic value for the entirety of reset-simulation and verification.Here is an example where the create_symbolic_value command is used to create a symbolic-value of width 4, which is then used to constrain the opcode ports in both the specification and implementation designs.

```
create_symbolic_value -bitwidth 4 -name op_sym

set_constant -spec -module alutop -value op_sym opcode
set_constant -impl -module alutop -value op_sym opcode
```

Note that during the reset-sequence, a port constrained using a value created by create_symbolic_value gets driven with Xs. This behavior can be overridden by specifying a concrete-value to be driven during the reset-sequence using the -resetval option. In the following example, 0s are driven during reset-sequence into spec.in2 while Xs are driven into spec.in1.

```
create_symbolic_value -bitwidth 4 -name op_sym
set_constant -spec -module alutop -value op_sym in1

create_symbolic_value -bitwidth 4 -name op_sym2 -resetval 0
set_constant -spec -module alutop -value op_sym2 in2
```

create_transactor

Instantiates a transactor module, which could be either combinational or sequential, to constrain a set of input ports or transform output ports.

Usage

create_transactor [-name *name*] {**-spec** | **-impl**}
[-defparam {{*param1 val1*} {*param2 val2*}...}] **-module *module port_description***

Arguments

- -name *name*

  Specifies the instance name of the transactor to be created. On successful completion of the create_transactor command, this name is returned.

- **-spec** | **-impl**

  Specifies whether the constraint is to be associated with the specification design or the implementation design.

- -defparam {{*param1 val1*} {*param2 val2*}...}

  Specifies the list of parameter-value pairs to override for this instantiation of the transactor.

  Note:
  This argument is now deprecated. Specify -defparam with the build_design command as shown in Examples 1 and 3 below.

- **-module *module***

  Specifies the name of the module from the constraints library to be used to create the transactor instance.

- ***port_description***

  Specifies the port connections of the module instance in Verilog style, either as an ordered port list or a list of named port connections.

Description

The create_transactor command instantiates a transactor module in the specification or implementation design connected in accordance with the port description. The transactor module must have been previously read into the constraints library using the build_design -cons command. Input ports constrained by transactors cannot be otherwise constrained or mapped.

If a transactor constrains an input port of one design which is implicitly mapped (using name-based rules) to a port with the same name in the other design, then SLEC automatically unmaps them. For example:

```
create_transactor -spec -name t1 \
   -module MY_INV_TRANS {.in(Spec_In_Inv) .out(spec.in)}
```

Here, the waveform Spec_In_Inv is being inverted to constrain spec.in. If the mapping between spec.in and impl.in still exists, it will result in incorrect stimulus being applied during the verification process; therefore SLEC automatically unmaps them.

All inputs of the transactor module which are left hanging or associated with an explicit waveform in the transactor instantiation are promoted to being input ports of the corresponding specification/ implementation design.

```
create_transactor -impl -name d1 \
   -module DELAY_THREE_CYCLES {.clk() .in() .out(impl.in) }
```

In this case, SLEC performs the following steps:

- Instantiate the transactor module DELAY_THREE_CYCLES in the implementation design.
- Remove the input port in from the interface of the implementation design.
- The signal in remains in the design and is driven by the output of the transactor.
- Add the input ports corresponding to clk and in to the interface of the implementation design
- Drive the inputs clk and in of the transactor from the newly introduced primary inputs corresponding to clk and in respectively.

To get a list of the newly created ports, use the get_transactor_port command. For example,

```
get_transactor_port -impl -instname dl -portname clk
```

The input ports created above can subsequently be used with other SLEC commands. For example:

```
set_clock_root -impl [get_transactor_port -impl \
   -instname dl -portname clk] -clock CLK
create_map -input spec.in [get_transactor_port -impl -instname dl \
   -portname in]
```

To obtain the hierarchical name of a flop in the instantiated transactor, the following sequence of commands can be used (in this example, the name of the transactor flop whose complete hierarchical name needs to be queried is delay_flop).

```
set transactor_inst [create_transactor -impl -name d1 -module DELAY \
 {.clk() .in() .out(impl.in) }]
set transactor_flop "spec.$transactor_inst.delay_flop"
```

Transactors can also be used to transform one or more output ports to obtain waveforms which can then be used in subsequent output-maps. A waveform is implicitly created when the output of a transactor is associated with a non-existent port. For example, in the create_transactor command below, the waveform w_one_hot is implicitly created and then used in an output-map:

```
create_transactor -spec -name onhot_inst \
   -module onehot {.in(spec.out) .out(w_one_hot)}
create_map -output w_one_hot impl.out
```

Restrictions

SLEC places the following restrictions on transactors:

- The transactor must have been previously read in to the constraints library using the build_design -cons command.
- Inputs of transactors can only be connected to waveforms or ports in the following categories. Note that for the examples presented in these categories, the transactor formal port inval is an input port and the formal port outval is an output port.
  - Untransformed input-ports of the corresponding design.

```
create_transactor -spec -name t1 \
   -module z_encoder { .inval(spec.opcode) \
   .outval(spec.encoded_opcode) }
```

  - Temporal transformation of an input-port in the corresponding design. For example:

```
create_waveform -name opcode_sampled -sample_start 1 spec.opcode
create_transactor -spec -name t2 \
   -module z_encoder { .inval(opcode_sampled) \
   .outval(spec.sampled_encoded_opcode) }
```

  - An untransformed output-port in the corresponding design. For example:

```
 create_transactor -spec -name t3 \
   -module encoder_outputs { .inval(spec.data_out) \
   .outval(w_encoded_output) }
```

  - An explicit-waveform or its temporal transformation. For example:

```
create_waveform -name only_four_or_five \
   -bitwidth 8 {(4|5) +}
create_transactor -spec -name t4 \
   -module onehotmod { .inval(only_four_or_five) \
   .outval(spec.in1) }
```

- Transactors cannot be connected to ports in both the specification and implementation designs.

Examples

Example 1

Assume the following Verilog module is read into the constraints library:

```
module mod256(out, in);
    parameter WIDTH=32;
    output [WIDTH 1:0] out;
    input [WIDTH 1:0] in;
```

```
    assign out = in[7:0];
endmodule
```

Specifying the following command sequence will instantiate a transactor with the instance name c_low, where the parameter WIDTH is overridden with the value 8. After the transactor gets instantiated, the input port spec.in2 is driven by the output of the transactor which is driven by spec.in1.

```
build_design -spec -top mod256 -defparam { {WIDTH 8} } constraints.v
create_transactor -spec -name c_low -module mod256 {spec.in2 spec.in1}
```

Example 2

Connects the input transactor to the ports InA and InB in the specification design to ensure that the ports are never at logic 1 at the same time. In this example, because the new input port names are based on the -name option, create_transactor will not automatically map the ports.

```
build_design -cons input_trans.v
create_transactor -spec -name spec_trans -module input_trans\
   {.out1 (spec.InA)\
    .out2 (spec.InB)
    .in1()
    .in2() }
```

Here, the Verilog code for the module input_trans is as follows:

```
module input_trans (in1, in2, out1, out2);
    input in1, in2;
    output out1, out2;

    assign out1= in1 & ~in2;
    assign out2 = ~in1 & in2;
endmodule
```

Note:
The new input ports to the design become the transactor input ports in1 and in2.

Example 3

The following example demonstrates the use of a reconfigurable stall transactor, stall_xtor. This synchronous verilog module implements a clock period stall to identify when a signal is enabled high. The stall throughput is reconfigurable via a defparam THROUGHPUT parameter setting. The command sequence below defines two stall transactors, each with distinct clock throughput settings.

```
set stall2cycs \
 [ build_design -cons -top stall_xtor \
 -defparam {{ THROUGHPUT 2 }} stall_xtor.v ]

set stall4cycs \
 [ build_design -cons -top stall_xtor \
 -defparam {{ THROUGHPUT 4 }} stall_xtor.v ]
```

```
create_transactor -impl -module $stall2cycs  -name xtor_2cycles \
 {.clk() .signal(), .stalled_signal(impl.ready2)}

create_transactor -impl -module $stall4cycs  -name xtor xtor_4cycles \
 {.clk(), .signal(), .stalled_signal(impl.ready4)}
```

The command sequence above synchronizes active high signals on impl.ready2 and impl.ready4 every two and four clock cycles, respectively.

create_waveform

Creates a waveform by explicitly specifying a sequence of values or by transforming another waveform.

Usage

create_waveform [-name *name*] [-signed] [-bitwidth *bitwidth*] [-sample_freq *sample_freq*] [-sample_start *sample_start*] [-sample_end *sample_end*] [-hold_interval *hold_interval*] [-hold_start *hold_start*] [-hold_end *hold_end*] ***source_waveform***

Arguments

- -name *name*

  Name for the waveform being created. If not specified, a unique name is automatically generated. The name is returned as the result of the command.

- -signed

  For explicit waveforms, specifies that the type of this waveform is signed. Otherwise, the type is deemed to be unsigned.

- -bitwidth *bitwidth*

  For explicit waveforms, specifies a fixed bit-width type.

- -sample_freq *sample_freq* -sample_start *sample_start* -sample_end *sample_end*

  For fixed sampling waveform transformations, the frequency, start, and end parameters, which default to 1, 0, and infinity, respectively.

- -hold_interval *hold_interval* -hold_start *hold_start* -hold_end *hold_end*

  For holding waveform transformations, the interval, start, and end holding parameters, which default to 1, 0, and infinity, respectively.

- ***source_waveform***

  For explicit waveforms, a sequence of explicit waveform values. For waveform transformations, the source waveform.

Description

A waveform describes an infinite sequence of values of a specific type. The create_waveform command creates a waveform from explicit values or from waveform transformations.

Every port of the design has an implicit waveform associated with it which represents the sequence of values read or written by that port. In commands which use waveforms, the port name is used to refer to this waveform.

This command returns the name of the waveform created.

Explicit Waveforms

An explicit waveform is described using regular expressions. For example, this command defines a single bit waveform called RESET_ON_LOW, that is logic 0 for one cycle, then logic 1 for all cycles after that. By using the name RESET_ON_LOW, this waveform can be referenced in other commands as well. Note that the name is optional. If not specified, a unique name is assigned by SLEC and returned as the command's result.

```
create_waveform -name RESET_ON_LOW -bitwidth 1 {0 1+}
```

The following example creates a waveform with values 8 in the 1st cycle, 12 in 2nd cycle then 18 forever. In SLEC, "forever" means until the end of the transaction (as defined by throughput) or the end of the reset state if the waveform is applied during reset.

```
create_waveform -name WAVE1 -bitwidth 8 {8 12 18+}
```

Values in waveforms can be repeated. This example creates a waveform in which the value in the first cycle is symbolic (this can be any value in the 8-bit range), then has the value 44 for the next 8 cycles, and finally any value between 50 and 70 in every cycle after that.

```
create_waveform -name WAVE3 -bitwidth 8 {S 44[+8] (50-70)+}
```

Negative values can be specified only for signed explicit waveforms. A negative number needs to be specified within parenthesis. In the example below, the waveform WAVE4 has the value -2 in the first cycle, the value -1 in the second cycle, and the value 0 forever after that.

```
create_waveform -name WAVE4 -signed -bitwidth 4 { (-2) (-1) 0+}
```

All waveform sequences are infinite in length. 0+ at the end of the sequence means that 0 is to be repeated forever for waveform0. If the explicit sequence had a finite sequence of values, then SLEC appends an infinite sequence of symbolic values to complete the waveform. So the following waveforms are identical:

```
create_waveform -name waveform1a -bitwidth 1 { 1 0 1 0 }
create_waveform -name waveform1b -bitwidth 1 { 1 0 1 0 S }
create_waveform -name waveform1c -bitwidth 1 { 1 0 1 0 S+ }
```

Generally, values are specified using a Verilog-like syntax, for example:

```
42, 16'hCAFE, (-'d06712)
```

Simple decimal number (including digits 0 to 9) used to specify values must be in the range -2147483647 to 2147483647. Values outside of this range can be specified as based literal constants, without a sign designator (s or S). The following examples are valid specifications:

```
create waveform -signed -bitwidth 32 {(-2147483647) 0+}
create waveform -signed -bitwidth 32 {(-'d2147483648) 0+)}
```

The following specification is invalid because the number is not designated as a based literal constant, since it is out of the required range:

```
create_waveform -signed -bitwidth 32 {(-2147483648) 0+}
```

A subset of values can be described using the | and ^ operators. The | operator is used to specify a choice of values, and the ^ operator is used to specify any value except the value mentioned along with the operator. For example,

```
create_waveform -name odd_only -bitwidth 3 { (1|3|5|7)+ }
create_waveform -name one_non_zero -bitwidth 3 { ^0 0+ }
```

Another way to specify a subset of values is using the - operator. The - operator is used to specify a range of values. It can be used in conjunction with any of the aforementioned operators. For example:

```
create_waveform -name tens_only -bitwidth 8 { (10-19)+ }
create_waveform -name anything_but_tens -bitwidth 8 { ^(10-19)+ }
```

Here, the waveform tens_only only takes values in the range 10-19 and the waveform anything_but_tens takes any possible 8-bit value except in the range 10-19.

Transformed Waveforms

A waveform can sample another waveform beginning in a start cycle repeating every frequency cycles until an end cycle using the following arguments:

```
-sample_freq <sample_freq> -sample_start <sample_start> -sample_end <sample_end>
```

Waveforms begin with index 0. The default sample parameters are a start of 0, frequency of 1, and end of infinity. For example,

```
create_waveform -name waveform3 -sample_freq 2 -sample_start 0 -sample_end 10 spec.out
```

creates a waveform that samples the waveform associated with the port spec.out starting with the value at index 0, every 2 values, up to index 10. The waveform is symbolic after that.

Consider another example, where the data can take the values 1, 2 or 3 in the first cycle and the second cycle can take any value in the 6-bit range except 14. In this example, since no other values are defined, the waveform is symbolic after the 2nd cycle.

```
create_waveform -name WAVE2 -bitwidth 6 {(0|1|3) ^14)}
```

A waveform can sample and hold another waveform beginning in a source start cycle holding that value for an interval and repeating until a source end cycle is reached using the following arguments:

```
{[ -hold_interval <hold_interval> ] [ -hold_start <hold_start> ] [ -hold_end <hold_end> ] }
```

For example,

```
create_waveform -name waveform6 -hold_interval 2 -hold_start 0 -hold_end 7 spec.in
```

creates a waveform which holds the first value of the spec.in waveform for two cycles, holds the next seven values of spec.in for two cycles each, and becomes symbolic after that.

If no starting cycle is specified, it defaults to cycle 0. If no interval is specified, it defaults to 1 cycle. If no ending cycle is specified, the waveform is sampled forever.

A note on the nomenclature of SLEC generated waveform names

While generating names for waveforms created using inlined Tcl commands i.e. ones which do not explicitly specify a waveform name, SLEC uses the convention of adding suffixes to the base-waveform being transformed according to the following conventions:

```
-hold_start <hold_start>          : _hs=<hold_start>
```

```
-hold_interval <hold_interval>           : _hi=<hold_interval>
```

```
-hold_end <hold_end>           : _he=<hold_end>
```

```
-sample_start <sample_start>           : _ss=<sample_start>
```

```
-sample_freq <sample_freq>           : _sf=<sample_freq>
```

```
-sample_end <sample_end>           : _se=<sample_end>
```

For example, the names of the waveforms generated while evaluating the create_waveform commands in the following create_map command are spec.inval_ss=3_sf=2 and impl.mode_ss=12, respectively.

```
create_map -input [create_waveform -sample_start 3 \
 -sample_freq 2 spec.inval] [create_waveform -sample_start 12 impl.mode]
```

create_zero

Creates a constant with a value of 0.

Usage

create_zero [**-spec** | **-impl**]

Arguments

- **-spec** | **-impl**

  Specifies whether the constant created is to be mapped to an output port in the specification design or the implementation design.

  Use the -spec option to map the constant to an output port in the implementation design. To map the constant to an ouput port in the specification design, use the -impl option.

Description

Specifies whether the constant created is to be mapped to an output port in the specification design or the implementation design.

Use the -spec option to map the constant to an output port in the implementation design. To map the constant to an ouput port in the specification design, use the -impl option.

The command is typically used with the create_map command to map output ports against a constant value. This example maps impl.invalid_data to the constant 0.

```
create_map .output impl.invalid_data [create_zero -spec]
```

disable_msg

Disables reporting of messages of a particular message ID.

Usage

disable_msg [-after *value*] ***message_id***

Arguments

- -after *value*

  Number of times to display a message before disabling further reporting of the message. The default is 0, which will immediately disable further reporting.

- ***message_id***

  Message ID.

Description

The disable_msg command immediately turns off reporting for a specific message ID. For example, the following command prevents the display of the message ID ABC-DEF.

```
slec> disable_msg ABC-DEF
[UI-MID]    Message "ABC-DEF" is now disabled.
```

If the -after option is specified, reporting for the message ID is disabled after it has been reported by the number of times defined by the -after option. For example, the following command disables the display of message ID ABC-DEF after it has been issued 5 times.

```
disable_msg ABC-DEF -after 5
```

enable_msg

Enables reporting for messages previously disabled using the disable_msg command.

Usage

enable_msg [*message_id*]

Arguments

- *message_id*

  Message ID. If specified, messages with the specific ID are enabled. Otherwise, messages of all ID types are enabled.

Description

The enable_msg command is used to turn on reporting for a previously disabled message ID. For example, the following command enables display of the message ID ABC-DEF after display of the message has been disabled using the disable_msg command.

```
slec> enable_msg ABC-DEF
[UI-MIE]    Message "ABC-DEF" is now enabled.
```

exit

Terminates SLEC and returns control to the operating system shell.

Usage

exit [*exit_code*]

Arguments

- *exit_code*

  Specifies the value returned by SLEC on exit. If not specified, SLEC returns an exit code of 0. The exit code can be viewed using the UNIX command echo $?.

Description

The exit command causes SLEC to terminate and return control to the operating system shell. The exit code can be viewed using the UNIX command echo $?.

Examples

Example 1: Terminates SLEC and returns an exit code of 1.

```
slec> exit 1
0 warning, 0 error message
0.485u 0.166s 0.651 0:05.000          15.144m 29.219p
(SLEC process used 29 MB and 1 seconds)
[/home/calypto]$ echo $?
1
```

Example 2: Terminates SLEC and returns the default exit code, 0.

```
slec> exit
0 warning, 0 error message
0.500u 0.171s 0.671 0:04.000          22.509m 41.375p
(SLEC process used 41 MB and 1 seconds)

[/home/calypto]$ echo $?
0
```

find

Queries the design database for information about modules, ports, signals, and instances in the designs.

Usage

find { **-module** | **-inst** | **-port** | **-signal** } [-spec | -impl] [-blackbox | -sequential] [-bind] [-input | -output | -inout | -clock] [-user] [-hier] [-short_name | -count] [-wildcard | -exact | -regexp] [*expr*]

Arguments

- **-module** | **-inst** | **-port** | **-signal**

  Searches for modules, instances, ports or signals.

- -spec | -impl

  When searching for instances, ports or signals, restricts search to the specification or implementation design.

- -blackbox

  When searching for instances, restricts search to blackboxed instances.

- -sequential

  When searching for instances, restricts search to sequential (flops, latches, and memory) elements.

- -bind

  When searching for instances, restricts search to instances added through SystemVerilog bind directives. To find all instances in a design, use the following:

  ```
  find -hier -inst -bind
  ```

- -input | -output | -inout | -clock

  When searching for ports, restricts search to primary or black box input, output, inout or clock ports.

- -user

  Restricts search to objects present in the original design. Any additional objects generated by SLEC during design interpretation and optimization are not included.

- -hier

  Normally, the search pattern is applied at the top of the instance hierarchy. When -hier is specified, the search pattern is applied to every hierarchical instance as well. This option has no impact when searching for modules.

- -short_name | -count

  Specifies the type of information to be returned, either object names or a count (-count) of the number of objects found. By default, original hierarchical path names are returned. When the -short_name option is specified, short names are returned so hierarchical path information is not displayed.

- -wildcard | -exact | -regexp

Interprets the search expression using wildcard, exact, or regular expression matching rules. Wildcarding is the default matching style. Wildcarding extends UNIX shell * and ? glob matching with a ** operator, which matches zero or more levels of hierarchy.

- *expr*

  A search expression which names a set of objects in the designs. The default search expression is "*". Use braces around the expression, {<*expr*>}, when part-selecting or using escaped names that include bracket or backslash caracters ([,], or \ ).

Description

The find command searches the database to find information about modules, instances, ports, or signals that match the given search criteria. Other commands that accept hierarchical names to access design objects use this command internally, so all the naming rules which apply to this command also apply to those commands.

There are four types of information required for each search:

- Kind of object to search
- Scope of the search
- Output format
- Search expression

Kind of Object

This command searches for modules, instances, ports and signals.

Instance searches can be restricted to sequential (-sequential) or blackboxed (-blackbox) objects. Sequential objects are explicitly specified or implicitly inferred flops, latches, and memory elements present in a design. Black boxes are all module instances blackboxed by the user.

When the -user option is specified, search is restricted to objects present in the original design; objects generated during module optimizations are not included.

Scope of the search

The scope of the search must be restricted to the specification or implementation design.

Normally, the search pattern is applied from the top of the instance hierarchy. When the -hier option is specified, the search pattern is applied to every hierarchical instance in the design. For example, the following command will find a port_A port at the top instance of the specification design:

```
find -spec -port port_A
```

On the other hand, specifying the following command will find port_A ports of all instances in the specification design.

```
find -spec -hier -port port_A
```

Output format

A search result is returned as a list or count of objects found. If no object matches, an empty list or zero count is output. There are 3 output formats:

default	original hierarchical names
-short_name	names without hierarchical information
-count	count of objects found

The default is by list of original hierarchical names.

Search expression

A search expression matches the object to be described. The search expression can be interpreted using wildcard, exact, or regular expression semantics. The default is wildcard semantics.

See the Appendix for more details on wildcard and regexp matching rules.

The search expression for -instance, -port and -signal is the hierarchical name of that object. The search provides objects in a uniquified/flattened view of the module. For example:

```
find -port -spec i5.i2.*
```

returns all ports of instance i5.i2 in the specification design:

```
 i5.i2.p0 i5.i2.p1
```

Note that * wildcarding will not cross levels of hierarchy. In the previous example, ports of instances below i5.i2 will not be found. Instead, use ** wildcarding:

```
find -port -spec i5.**.p*
```

returns all ports beginning with p which are in instance i5 or any instances contained in i5:

```
i5.p3 i5.i1.p0 i5.i2.p0 i5.i2.p1 i5.i2.i3.p0 i5.i2.i3.p2
```

The search expression for a module is the name of the module only. For example:

```
find -spec -module mod_a*
```

This returns all the modules in the specification design with mod_a as the prefix:

```
mod_adder mod_addmult mod_alu
```

Some object types, such as memories and data structures, might be flattened by SLEC. While searches will use the original names, flattened object names will be returned.

For example, memory arrays get blasted into registers in the database. The registers have the blasted names. The following examples shows all the kind of finds that can be performed. Assume the following memories are instantiated in the designs:

```
reg [7:0]        mem[3:0];          // Verilog

sc_in<sc_int<7> >       port_mem[1:0][1:0][1:0];          // SystemC
```

then:

```
slec> find -spec -inst mem
  mem[0], mem[1], mem[2], mem[3]

slec> find -inst mem\[2\]
  mem[2]

slec> find -spec -port port_mem
  port_mem[0][0][0] port_mem[0][0][1]
  port_mem[0][1][0] port_mem[0][1][1]
  port_mem[1][0][0] port_mem[1][0][1]
  port_mem[1][1][0] port_mem[1][1][1]

slec> find -spec -port {port_mem[1]}
  port_mem[1][0][0] port_mem[1][0][1]
  port_mem[1][1][0] port_mem[1][1][1]

slec> find -spec -port {port_mem[1][1]}
  port_mem[1][1][0] port_mem[1][1][1]

slec> find -spec -port port_mem\[1\]\[0\]\[1\]
  port_mem[1][0][1]
```

Structures are also flattened in the database during synthesis into the netlist. The following examples show the kinds of finds that can be performed. Assume the following SystemC structures are used in the designs:

```
struct addr {
 sc_int<8> local;
 sc_int<4>  region;

struct pkt {
 addr  addr1;
 sc_int<7> data[2][2];
};

sc_signal<pkt> in_pkt;
```

then:

```
slec> find -spec -signal in_pkt
  in_pkt.addr1.local in_pkt.addr1.region in_pkt.data[0][0]
  in_pkt.data[0][1] in_pkt.data[1][0] in_pkt.data[1][1]
slec> find -spec -signal in_pkt.addr1
  in_pkt.addr1.local in_pkt.addr1.region
slec> find -spec -signal in_pkt.addr1.data\[0\]
  in_pkt.addr1.addr1.data[0][0] in_pkt.addr1.addr1.data[0][1]
```

Because SLEC tries to work at the word level, if a design depends on an object, but not on any specific bit or part of that object, then SLEC might not flatten the object into its bit level representation. In that case, only the object itself might be searched for and not a specific bit or part select of the object.

Signal Aliasing

When SLEC builds a design database, some limited optimizations are done on logic inside design modules. As a result of these optimizations, some internal signals might be optimized away, while other internal signals might be merged because they have the same functionality. Ports will not be affected.

When a signal is optimized away, it will never appear in a search result list.

If two or more signals become merged, one of the original signal names is chosen to be the name of the merged signal. Other signal names become aliased to that name. Searching for any one of the original signal names will always return the name of the merged signal.

For example, consider that signals a and b in the spec instance are merged together, with signal a becoming the name of the merged signal. A search for either signal will return signal a:

```
SLEC>find -signal spec.a
  spec.a

SLEC>find -signal spec.b
  spec.a
```

findlist

Queries the design database for information about modules, ports, signals, and instances in the designs, building a results list for other commands.

Usage

findlist { **-module** | **-port** | **-signal** | **-inst** } [-spec | -impl] [-blackbox | -sequential] [-input | -output | -clock] [-user] [-hier] [-require *require* ...] [-append *append* ...] [*expr*]

Arguments

- **-module** | **-port** | **-signal** | **-inst**

  Specifies the type of object to find. Supported design objects are modules, instances, ports and signals.

- -spec | -impl

  If searching for instances, ports or signals, restricts search to the specification or implementation design.

- -blackbox

  When searching for instances, restricts search to blackboxed instances.

- -sequential

  When searching for instances, restricts search to sequential (flops, latches, and memory) elements.

- -input | -output | -clock

  When searching for ports, restricts search to primary or black box input, output, or clock ports.

- -user

  Specifies that search is to be restricted to objects present in the original design; any additional objects generated by SLEC during design interpretation and optimization are not included.

- -hier

  Normally, the search pattern is applied at the top of the instance hierarchy. If -hier is specified, apply the search pattern to every hierarchical instance as well. Has no effect if searching for modules.

- -require *require* ...

  One or more search expressions which will name a set of objects in the designs which are required to be present to complete the match. For each value matched in the primary search expression, wildcard substitution parameters, %1-%9, if present, are replaced in the required search expressions. To complete a match, the primary search expression value and corresponding values in all required search expressions must be present in the database.

- -append *append* ...

  One or more expressions representing values to append to the return list if a match is found. For each value matched in the primary search expression, wildcard substitution parameters %1-%9, if present, are replaced in the appending expressions.

- *expr*

The primary search expression which names a set of objects in the designs using wildcard substitution. Wildcarding extends UNIX shell * and ? glob matching with a ** operator which matches zero or more levels of hierarchy. The default search expression is "*". If a match is found, the primary value and corresponding required and appending values are appended to the list returned by the command.

Description

The findlist command searches the database to find modules, ports, signals, and instances that match the given search criteria. Other commands that accept hierarchical names to access design objects use this command internally, so all the naming rules which apply to this command also apply to those commands.

Consider the following example. This example first shows the specification and implementation ports returned by the find command, and then subsequently uses the findlist command to list the specification ports that have matching ports in the implementation design.

```
slec> find -port spec.*
spec.In1 spec.In2 spec.Out1

slec> find -port impl.*
impl.In1 impl.In2 impl.Out1 impl.Out2

slec> findlist -port spec.* -require impl.%1
spec.In1 impl.In1 spec.In2 impl.In2 spec.Out1 impl.Out1
```

In this example, the findlist command does not return the implementation port Out2 since there is no matching port in the specification design.

Note that the findlist command can be used to match multiple parameters. The first specification wildcard is matched to the first implementation wildcard (%1) and the second wildcard in the specification design, matched to the second wildcard (%2) in the implementation design. For example,

```
findlist -inst spec.*.U1.reg* -require impl.%1.U1.reg%2
```

Consider another example. To find all ports which have the same hierarchical path but start with p in spec and p_ in impl, use the following command.

```
findlist -port spec.**.p* -require impl.%1.p_%2
```

which returns:

```
spec.p0 impl.p_0 spec.i0.p0 impl.i0.p_0 spec.i0.p1 impl.i0.p_1
 spec.i0.i1.i2.i3.i4.p1 impl.i0.i1.i2.i3.i4.p_1
```

Refer to the find command for more information and examples of search parameters.

find_nodes

Returns the flops found in the fanin or fanout cone of a given list of signals.

Usage

find_nodes **-fanin** | **-fanout** {**-start_points** ***start_point_list***} { **-depth** ***depth*** | **-end_points** ***end_point_list*** } [-hier]

Arguments

- **-fanin** | **-fanout**

  Specifies whether the flops to be searched for are located in the fanin or fanout cone.

- **-start_points** ***start_point_list***

  Specifies the list of starting signals. The fanin or fanout cones of all the signals specified in the starting signal list are returned.

- **-depth** ***depth***

  Specifies the sequential depth for the fanin/fanout traversal. A depth of 1 implies that the traversal should stop at the first flop boundary. This option cannot be used in conjunction with -end_points.

- **-end_points** ***end_point_list***

  Specifies the list of ending signals. While tracing the fanin or fanout cone, the search for a flop is made until an end point is reached or a primary input, primary output or a black box port is reached. This option cannot be used in conjunction with the -depth option.

- -hier

  By default, the search pattern is applied at the top of the instance hierarchy. If -hier is specified, the search pattern is applied to every hierarchical instance as well.

Description

The find_nodes command searches for flops in the fanin or fanout cone of a given list of signals and returns the list of flops found. The result is the union of all the flops, found by tracing the fanin or fanout cone of every start point specified in the start signal list until one of the end points, a primary input/output port, a black box port or the specified sequential depth is reached. If a flop is in the fanin or fanout cone of multiple start points, it is listed only once.

To illustrate how the find_nodes command works, consider the following Verilog module as the specification design.

```
module top(out, clk, d1, d2, reset);
output out;
input clk;
input d1, d2;
input reset;
reg fl1;
reg fl2;
reg fl3;
always @(posedge clk)
    begin
```

```
    if (reset == 1'b1)
        fl1 <= 0;
        fl2 <= 0;
        fl3 <= 0;
    else
        fl1 <= d1;
        fl2 <= d2;
        fl3 <= fl1;
    end
    assign out  = !fl3 | fl2;
endmodule
```

In the following example, find_nodes reports all the flops in the fanin cone of signal spec.out until a sequential depth of 1.

```
SLEC>find_nodes -fanin -start_points "spec.out"  depth 1
```

This command returns:

```
spec.fl3 spec.fl2
```

To return the flops which exist in the fanout cone of spec.d1 and also in the fanin cone of spec.out.

```
SLEC>find_nodes -fanout -start_points "spec.d1" -end_points "spec.out"
```

This command returns:

```
spec.fl1 spec.fl3
```

Examples

To illustrate the use of the hier and depth options, consider the following examples.

This example shows all the flops in the fanout from instance impl.U04.q_reg0 to output port impl.Out3.

```
find_nodes -fanout -start_points impl.U04.q_reg0 -end_points impl.Out3 \
   -hier impl.U04.q_reg1 impl.U04.q_reg2 impl.U04.q_reg3 impl.U04.q
```

Without the hier option, the list returned from find_nodes is limited to the hierarchy of the start point. Using the same design as above, this example lists all the fanout flops in the same level of hierarchy and displays a warning that the end point is not reachable.

```
find_nodes -fanout -start_points impl.U04.q_reg0 \
        -end_points impl.Out3
<WRN> [CDB-FEPNT] End point "impl.Out3" not reachable in the "fanout"
        impl.U04.q_reg1 impl.U04.q_reg2 impl.U04.q_reg3 impl.U04.q
```

The -depth option can be used to limit the depth reported. This is useful for finding all fanin flops when a conditional proof is given for a flop.

```
find_nodes -fanin -start_points impl.U04.q_reg0  -depth 2
impl.U04.q_reg0 impl.U04.b_Reg
```

get_async_subsession_error_code

Returns the error code of the asynchronous job with the specified job handle.

Usage

get_async_subsession_error_code **-handle *job_handle***

Arguments

- **-handle *job_handle***

  Specifies the handle of the asynchronous job whose error code must be returned.

Description

If the job handle is invalid, the command returns -2. If the job has not terminated yet, the command returns -1. Otherwise, it returns the error code for the corresponding job.

get_async_subsession_status

Returns the current status of the asynchronous job that has the specified job handle.

Usage

get_async_subsession_status **-handle *job_handle***

Arguments

- **-handle *job_handle***

  Specifies the handle of the asynchronous job whose current status must be returned.

Description

Returns an enumerated data type. Following are the available options:

- **JOB_QUEUED** : The job is queued.
- **JOB_RUNNING** : The job is running.
- **JOB_FINISHED** : The job has completed.
- **JOB_INVALID** : An invalid job handle was specified.

Note:
In Tcl, the enumerated type is returned as a string.

get_async_subsession_workdir

Returns the work directory that is currently running the task associated with the specified job handle.

Usage

get_async_subsession_workdir **-handle *job_handle***

Arguments

- **-handle *job_handle***

  Specifies the handle whose associated job is being run in the work directory.

Description

Returns the work directory. Returns "??" if the specified job handle is invalid.

get_async_subsession_worker

Returns the handle to the worker process that is currently running the task associated with the specified job handle. This is valid for jobs that are in the RUNNING state.

Usage

get_async_subsession_worker **-handle** ***job_handle***

Arguments

- **-handle** ***job_handle***

  Specifies the handle whose associated job is being run by the worker process.

Description

The get_async_subsession_worker command returns a valid handle to the worker process. The command returns ??, if the job is not in the RUNNING state or if the specified job handle is invalid.

get_async_worker_job

Returns a handle to the asynchronous job that is currently being run by the worker process. This API works inverse of the get_async_subsession_worker API.

Usage

get_async_worker_job **-handle** ***worker_handle***

Arguments

- **-handle** ***worker_handle***

  Specifies the handle of the worker process that is running the asynchronous job.

Description

The command returns the handle to an asynchronous job. Returns -1 if the worker process is not running.

get_async_worker_status

Returns the internal state of the worker process that has the specified handle.

Usage

get_async_worker_status **-handle *worker_handle***

Arguments

- **-handle *worker_handle***

  Specifies the handle of the worker process.

Description

This command returns an enumerated type. Following are the available options:

- **RUNNING** : Specifies that the worker process is running a job.
- **AWAITING_CONNECT** : Specifies that the master process is waiting for LSF to start the worker process and for the worker process to ping home.
- **AWAITING_ID** : Specifies the startup state, where the master process is waiting for the worker process to report its hostname and configuration.
- **AWAITING_CHDIR** : Specifies the startup state, where the master process is waiting for the worker process to change its work directory.
- **IDLE** : Specifies that the worker process is idle. Although SLEC is not running, the worker process is active on a remote machine and might be consuming LSF resources, depending on how the LSF has been setup.
- **DIED** : Specifies that the worker process has died. This can happen only if any of the following options is true:
  - The socket was closed.
  - The worker process resulted into an error state and, therefore, was killed.
  - You ran the shutdown_async_worker API.

Note:
Returns DIED if the specified worker handle is invalid.

get_async_worker_time_elapsed

Returns the time duration that has elapsed since the last state transition happened for the worker with the specified handle. This can be used to determine the duration for which a job has been running or a worker process has been idle.

Usage

get_async_worker_time_elapsed **-handle *worker_handle***

Arguments

- **-handle *worker_handle***

  Specifies the handle of the worker process.

Description

The command returns the number of seconds. Note that the data-type is double.

Returns 0 if the specified worker handle is invalid.

get_global

Returns a global variable value.

Usage

get_global ***name***

Arguments

- ***name***

  Name of a global variable.

Description

Returns the value of a global variable. For example, the following command returns the value of the global sim_max_transactions.

```
slec> get_global sim_max_transactions
100
```

Refer to the global variable reference chapter for a list of currently supported global variables.

get_signal_width

Returns the bit-widths of signals matching the argument name.

Usage

get_signal_width ***signal_name***

Arguments

- ***signal_name***

  Name of the signal(s) for which to compute bit-width.

Description

The get_signal_width command calls the find command to generate a list of signals whose names match signal_name and returns a list of bit-width values in the same order as the generated signal list. If the signal list is empty, the command returns an error.

get_transactor_port

Returns a list of the port(s) created in the specification/implementation design for the specified transactor instance.

Usage

get_transactor_port {**-spec** | **-impl**} **-instname** ***name*** [-portname *name*]

Arguments

- **-spec** | **-impl**

  Specifies whether the specification design or the implementation design contains the transactor instance.

- **-instname** ***name***

  Specifies the instance name of the transactor returned by the create_transactor -name command.

- -portname *name*

  Specifies the name of the formal port in the transactor module for which the corresponding port created (if any) in the specification/implementation design should be returned.

Description

The get_transactor_port command works in conjunction with the create_transactor command.

The create_transactor command instantiates the specified module in the specification/implementation design; it also creates additional design input/output ports corresponding to unconnected formal ports in the transactor module.

The get_transactor_port command, on the other hand, can be used to obtain their names for use in subsequent commands. If no ports are created for the specified transactor instance, then a null list will be returned. In case, the -portname option is used and no design port was created corresponding to the specified port, then no port will be returned.

This example shows how the get_transactor_port command is used to retrieve the new input port names created by the create_transactor command. The example further shows how the create_map command can be used to map the transactor input ports using their new names.

```
# Connect the transactor to the specification and implementation ports
create_transactor -spec -name spec_trans -module input_trans \
       { .out1(spec.Ch1a) \
         .out2(spec.Ch1b)
         .in1 ()
         .in2 () }
create_transactor -impl -name impl_trans -module input_trans \
       { .out1(impl.Ch1a) \
         .out2(impl.Ch1b)
         .in1 ()
         .in2 () }
# Get the port names and assign to Tcl variables
set spec_in1 [get_transactor_port -spec -inst spec_trans -port in1]
```

```
set spec_in2 [get_transactor_port -spec -inst spec_trans -port in2]
set impl_in1 [get_transactor_port -impl -inst impl_trans -port in1]
set impl_in2 [get_transactor_port -impl -inst impl_trans -port in2]
# Use the new port names to map the new input ports
create_map -input $spec_in1 $impl_in1
create_map -input $spec_in2 $impl_in2
```

get_version

Returns the version number of the SLEC release.

Usage

```
get_version [ -major | -minor | -patch | -update ]
```

Arguments

- -major

  Returns the major release number.
- -minor

  Returns the minor release number.
- -patch

  Returns the patch release number.
- -update

  Returns the update release number.

Description

The get_version command returns the SLEC version number or a specified field thereof. Depending on the SLEC release date, the version number follows one of two schemes:

- Releases 10.5 and prior have a version number of the form M.NU_P, where M is the major (numeric), N is the minor (numeric), U is the update (alphabetical), and P is the patch (1, 2, 3, RC<n>, NIGHTLY, and so forth). U and P are optional. As an example, release number 10.4a_1 has the major number is 10, the minor number 4, the update letter a, with the patch number 1.

- Releases after 10.5 have a version number of the form YYYY.N_P, where YYYY is the major (indicating release year), N is the minor (1, 2, 3, and so forth, indicating the release number for the year), and P is the patch (1, 2, 3, RC<n>, NIGHTLY, and so forth). P is optional, and there is no update field. As an example, release number 2021.2_3 has the major number 2021, the minor number 2, and the patch number 3.

help

Provides command shell help.

Usage

help [commands | *command* | globals | *global* | *message_id*]

Arguments

- commands

  List all user commands.
- *command*

  Describe a specific user command.
- globals

  List all user global variables.
- *global*

  Describe a specific user global variable.
- *message_id*

  Describe a specific message ID.

Description

The help command provides the user with help about commands, global variables, and messages.

For example,

```
help build_design
```

returns help for the build_design command.

insert_bbox_and_assume_guarantee

Inserts a black box on a specified signal and sets an assume guarantee framework around it that checks if the constraints specified on the signal are true. Returns the instance name of the inserted black box which can then be mapped using the map_assume_guarantee command.

Usage

insert_bbox_and_assume_guarantee [-onehot | -range -min *min_value* -max *max_value* | -constant *value*] ***signal***

Arguments

- -onehot | -range -min *min_value* -max *max_value* | -constant *value*

  Use the -onehot option to specify that the signal will always be a onehot value in the design. For example, for a 2-bit signal, the values would always be 01 or 10.

  Use the -range option to specify that the signal will always assume a value within the range specified. This option is particularly useful when a signal can assume a wide range of values but given the constraints applied to the design, the signal will never assume all values but just a subset of it.

  Use the -constant option to specify that the value of the signal will remain constant.

- ***signal***

  Signal on which the black box is to be inserted.

Description

The insert_bbox_and_assume_guarantee command inserts a black box on the specified signal, returns its instance and sets an assume guarantee framework around it that:

- assumes that the constraints specified on the black box using the above mentioned options hold true for the black box output (primary input).
- guarantees that the same constraints hold true on the input of the black box (primary output).

Examples

Example 1: Blackboxes the 3-bit signal sig1 and sets an assume guarantee framework around it that checks that the signal sig1 is a onehot signal that only assumes the values 001, 010 or 100.

```
insert_bbox_and_assume_guarantee  -onehot sig1
```

Example 2: Assuming the signal sig2 can assume a value between 0 and 15 but given the constraints applied to it, the signal will never assume a value greater than 5. Specifying the following command will insert a black box on the signal sig2 and set an assume guarantee framework around it that checks if the constraints specified on the signal are true.

```
insert_bbox_and_assume_guarantee -range -min 0 -max 5 sig2
```

Example 3: Blackboxes the signal sig3 and sets an assume guarantee framework around it that checks that the signal sig3 always assumes a constant value of 4.

```
insert_bbox_and_assume_guarantee -constant 4 sig3
```

Example 4: Assuming the signal spec.state.ff_count and impl.state.ff_count can assume any value but given the constraints applied to them, the signals will never assume a value greater than 60. By specifying the insert_bbox_and_assume_guarantee command, a black box and assume guarantee framework is set around each signal that checks if the constraints specified on the signal are true. The names of the black boxes are stored in the variables ff_count_spec and ff_count_impl for the specification and implementation designs respectively. The resultant black boxes are then mapped using the map_assume_guarantee command.

```
set ff_count_spec [insert_bbox_and_assume_guarantee -range \
      -min 0 -max 60  spec.state.ff_count]
set ff_count_impl [insert_bbox_and_assume_guarantee -range \
      -min 0 -max 60  impl.state.ff_count]
map_assume_guarantee $ff_count_spec $ff_count_impl
```

insert_black_box

Inserts a black box on a specified signal and returns its instance name. The inserted black box can be used in downstream constraints/maps like a regular black box.

Usage

insert_black_box **-signal *signal*** [-inst *inst*]

Arguments

- **-signal *signal***

  Specifies the signal on which the black box is to be inserted.

- -inst *inst*

  Specifies the instance name to use for the inserted black box. By default, SLEC uses the name <*signal_name*>_insert_bbox.

Description

The insert_black_box command is used to insert a black box on the specified signal. By default, the instance name of the inserted black box is <*signal_name*>_insert_bbox. If this name conflicts with an existing instance within the scope of the signal, SLEC creates a unique name by adding a suffix. If the -inst option is specified, SLEC uses the specified name and issues an error in case of a conflict.

The black box will have an input port called bbox_in and an output port called bbox_out. The signal on which the black box is inserted will drive the input port of the black box node. The output port of the black box will drive a signal whose fanouts will be the ports which were earlier driven by the signal on which insert_black_box was performed. The newly created black box can be used in downstream constraints/maps.

Note:
Make sure that the constraint placed on the black box ouput, now the primary input of the design, is the same as the verified black box input to ensure that the proofs for maps involving downstream outputs are valid. Incorrectly constraining the black box output can result in a false-positive or a false-negative.

Examples

The following command can be used to insert a black box at a signal spec.b1.cpuout.

```
insert_black_box -signal spec.b1.cpuout
```

If there is no name conflict in the scope spec.b1, SLEC will create a black box with the instance name spec.b1.cpuout_insert_bbox. Signal spec.b1.cpuout will be connected to the input port of the black box and the signal spec.b1.cpuout_insert_bbox_out will be connected to the output port of the black box. This signal will now drive the logic which was earlier driven by the signal spec.b1.cpuout. The input port of the inserted black box will be called bbox_in and the output port of the black box will be called bbox_out

The example below shows how to insert black boxes at the specified signals, record the names of the inserted black boxes and map them.

```
set spec_bb [insert_black_box -signal spec.b1.cpuout]
set impl_bb [insert_black_box -signal impl.b2.cpuout]
create_black_box_map $spec_bb $impl_bb
```

is_async_worker_local

Determines whether the worker process associated with the specified job handle is running on the local or remote machine.

Usage

is_async_worker_local **-handle *worker_handle***

Arguments

- **-handle *worker_handle***

  Specifies the job handle whose associated worker process is running on the local or remote machine.

Description

The command returns 1 if the worker process is running on the local machine. Returns 0 if the specified job handle is invalid.

join_async_subsessions

Runs all the asynchronous subsessions.

Usage

join_async_subsessions [-periodic_cb *periodic_cb_proc*] [-notify_started_cb *notify_start_proc*] [-notify_finished_cb *notify_finish_proc*]

Arguments

- -periodic_cb *periodic_cb_proc*

  Specifies the periodic callback procedure. The periodic callback procedure is run every 60 seconds and has the following arguments:

  - List of queued job handles
  - List of running job handles
  - List of finished job handles
  - List of killed job handles
  - List of all workers

  Example:

  ```
  proc PeriodicCB {queuedJobs runningJobs finishedJobs killedJobs \
   workers}
  ```

- -notify_started_cb *notify_start_proc*

  Specifies the "notify started" procedure. The "notify started" procedure indicates a job has started and takes the job handle as an argument.

  Example:

  ```
  proc NotifyStarted {handle}
  ```

- -notify_finished_cb *notify_finish_proc*

  Specifies the "notify finished" procedure. The "notify finished" procedure indicates a job has finished. It takes the job handle and return code as arguments.

  Example:

  ```
  proc NotifyFinished {handle returnCode}
  ```

Description

Runs all asynchronous jobs configured with the "run_subsession -async" command. Returns 0 if all jobs completed with error code 0; otherwise, returns 1. Acts as a synchronization barrier, returning after all

asynchronous jobs are complete. If a worker is killed while the command is running, the corresponding job is re-queued with a new working directory. In this case you will see multiple start messages for the same job, one for each time it launches. The messages will differ by the worker and workdir components.

For more details, refer to Subsession-level Distributed Processing in the *SLEC User's Manual*.

Examples

The following example shows the callbacks and relevant Tcl API functions.

```
for {set i 0} {$i < 20} {incr i} {

 # A human-readable, unique name is nice
 set name [my_get_name $i] ;

 # These are the actual options for the subordinate SLEC job
 set options [my_get_options $i];
 # Queue the job
 set handle [eval run_subsession  -async  -workdir my_$name $options] ;
 # Save the association of $handle to human-friendly name
 set handle2name($handle) $name ;
}

join_async_subsessions \
  -periodic_cb PeriodicCB \
  -notify_started_cb NotifyStartedCB \
  -notify_finished_cb NotifyFinishedCB

proc PeriodicCB { queuedJobs runningJobs
  finishedJobs killedJobs workers } {
   puts "Running Subsessions"
}

proc NotifyStarted {handle} {
     set name $handle2name($handle)
     set status [get_async_subsession_status  -handle $handle]
     set worker [get_async_subsession_worker  -handle $handle]
     puts "Task '$name' (job $handle) is now $status on worker $worker"
}

proc NotifyFinished {handle returnCode} {
     set name $handle2name($handle)
     set now [clock format [clock seconds]  -format "%m%d_%H:%M:%S"]
     puts "$now DONE  $name  rc=$returnCode"
     if {$returnCode != 0} {
   my_record_job_failure $name
     } else {
     my_record_job_success $name
     }
}
```

kill_async_subsession

Kills the asynchronous job that has the specified job handle.

Usage

kill_async_subsession **-handle *job_handle***

Arguments

- **-handle *job_handle***

  Specifies the handle of the asynchronous job that must be killed.

Description

Kills the asynchronous job that has the specified job handle. If the job is enqueued, it is removed from the queue. If the job is currently running, it is killed.

limit

Sets upper limits on resource usage for a SLEC run. When either of the specified resource-limits is exceeded, the SLEC run is terminated.

Usage

limit [-time *value*] [-memory *value*]

Arguments

- -time *value*

  Limits the real time usage for this SLEC run by this value. If no units are specified, SLEC assumes the units to be 'seconds'. You can optionally specify any of the following units (ns, us, ms, s, min, m, hours, h, days, or d). SLEC also accepts the HH:MM:SS format. Examples of acceptable values include: 45 (assumed to be seconds), 45s, 30min, 30m, 7hours, 7h, 2days, 2d, and 07:30:45.

- -memory *value*

  Limits the memory usage for this SLEC run to this value. The value can be a postive integer or floating point value greater than 50MB. If no units are specified, SLEC assumes the units to be 'MB'. You can optionally specify any of the following units (mb, MB, gb, GB). Examples of acceptable values include: 900 (assumed to be MB), 900mb and 2gb.

Description

The limit command sets upper limits on resource usage for a SLEC run. If no limits are specified, the limit command prints the current limit settings and the remaining quantity for the corresponding resource.

Consider the following examples. In this example, SLEC will exit if the time is more than 400 seconds or if the memory exceeds 55 Megabytes.

```
limit -time 400
limit -memory 55
```

In this example, time is limited to 1 hour 22 minutes and the memory limit set to 1.5 Gigabytes.

```
limit -time 1:22:00 -memory 1.5gb
```

Multiple limit commands can be used in the same SLEC run. The most recent command takes effect when multiple commands attempt to set the same limit. For example, if the following commands are issued, the time limit applicable will be 1:22:00.

```
limit -time 400
limit -time 1:22:00 -memory 1.5gb
```

link_design

Elaborates and links a design library.

Usage

link_design [**-spec** | **-impl** | **-cons**]

Arguments

- **-spec** | **-impl** | **-cons**

  Design library to link.

Verilog and VHDL arguments

- -top *top_module_or_entity*

  Specify the top module or entity of the design. The specified design hierarchy gets elaborated and linked. The rest of the design gets pruned.

- -defparam *list_of_parameter_value_or_generic_value_pairs*

  Specify list of parameter or generic value pairs to override default values for the parameters or generics of the top-level module or entity.

  For example, to override the value of parameters p1 and WIDTH in the top module of the specification design:

  ```
  read_design cpu.v
  link_design -spec -defparam { {p1 10} {WIDTH 16} }
  ```

Description

The link_design command elaborates and links design libraries to resolve the definitions of module instances before a design can be built.

```
link_design -spec
```

will link the specification design library.

Modules may be referred to within other module definitions. The active design library is searched during linking to resolve the reference. If the module definition is not found, an error occurs.

If reading a SystemC design into the specification library, use the link_design command to link the files. Follow this command with a build_db command to build the database. The following example illustrates this.

```
read_design -spec design.cpp sub_module.cpp
link_design -spec
build_db -spec
```

Multiple VHDL files can be read into different VHDL libraries in the SLEC implementation library and then linked. For example:

```
read_design -impl -library LIB1 entity1.vhdl
read_design -impl -library LIB2 entity2.vhd
read_design -impl top_entity.vhd
link_design -impl
```

list_properties

Returns a list of properties based on the criteria specified. Note that this command can only be called after the build_design command.

Usage

list_properties **-ovl** | **-prop** [-spec | -impl]
[-assert_always | -assert_one_hot | -assert_never | -abr | -abw | -asc | -cas | -dbz | -ise | -umr]

Arguments

- **-ovl** | **-prop**

  Specifies whether the properties are to be reported for for OVL instances or for user assert statements.

- -spec | -impl

  Specifies whether the properties are to be reported for instances in the specification design or for instances in the implementation design. When no option is specified, properties for all instances are reported.

- -assert_always | -assert_one_hot | -assert_never | -abr | -abw | -asc | -cas | -dbz | -ise | -umr

  Use one of these options to filter the type of properties reported. Note that the -assert_always, -assert_never and -assert_one_hot options require the -ovl option to be specified. For all other options, the -prop option must be specified.

  Specify -assert_always to list properties generated by SLEC for OVL instances of type assert_always.

  Specify -assert_never to list properties generated by SLEC for OVL instances of type assert_never.

  Specify -assert_one_hot to list properties generated by SLEC for OVL instances of type assert_one_hot.

  Specify -abr to list properties generated by SLEC for out-of-bounds array and bit-vector reads.

  Specify -abw to list properties generated by SLEC for out-of-bounds array and bit-vector writes.

  Specify -asc to list properties generated by SLEC for user assert statements.

  Specify -cas to list properties generated by SLEC for incomplete case statements.

  Specify -dbz to list properties generated by SLEC for divide by zero and modulo by zero operations.

  Specify -ise to list properties generated by SLEC for illegal shift operations.

  Specify -umr to list properties generated by SLEC for uninitialized memory reads.

Description

The list_properties returns a list of properties depending on the criteria specified. By default, all properties are listed.

Examples

Example 1: Lists properties generated by SLEC for user assert statements in the specification design.

```
list_properties -prop -spec -asc
```

Example 2: Lists properties for all OVL instances in the design.

```
list_properties -ovl
```

Example 3: Lists properties generated by SLEC for divide by zero and modulo by zero operations in the specification design.

```
list_properties -prop -spec -dbz
```

map_assume_guarantee

Maps black boxes returned by the insert_bbox_and_assume_guarantee command.

Usage

map_assume_guarantee [-spec_bbox_in_latency *spec_bbox_in_latency*] [-impl_bbox_in_latency *impl_bbox_in_latency*] [-spec_bbox_out_latency *spec_bbox_out_latency*] [-impl_bbox_out_latency *impl_bbox_out_latency*] ***bbox_1 bbox_2***

Arguments

- -spec_bbox_in_latency *spec_bbox_in_latency*

  Specifies the latency of the specification design for the output map corresponding to the black box input. Default is 0.

- -impl_bbox_in_latency *impl_bbox_in_latency*

  Specifies the latency of the implementation design for the output map corresponding to the black box input. Default is 0.

- -spec_bbox_out_latency *spec_bbox_out_latency*

  Specifies the latency of the specification design for the input map corresponding to the black box output. Default is 0.

- -impl_bbox_out_latency *impl_bbox_out_latency*

  Specifies the latency of the implementation design for the input map corresponding to the black box output. Default is 0.

- ***bbox_1***

  Specifies the name of the black box returned by the insert_bbox_and_assume_guarantee command that is to be mapped to <*bbox_2*>.

- ***bbox_2***

  Specifies the name of the black box returned by the insert_bbox_and_assume_guarantee command that is to be mapped to <*bbox_1*>.

Description

Maps black boxes returned by the insert_bbox_and_assume_guarantee command.

map_hierarchies

Maps interfaces of sub-block instances in the specification and implementation designs.

Usage

map_hierarchies [-spec_instance_name *spec_inst_name*] [-impl_instance_name *impl_inst_name*] [-spec_hierarchy_latency *spec_latency*] [-impl_hierarchy_latency *impl_latency*] [-spec_hierarchy_active_cycle *spec_cycle*] [-impl_hierarchy_active_cycle *impl_cycle*] [-valid_signal *signal_name*] [-valid_signal_delay *signal_latency*] [-isolate] [-debug] **-port_list *tcl_list***

Arguments

- -spec_instance_name *spec_inst_name*

  Specifies the hierarchical name of the instance created by SLEC corresponding to a function call in the C++ design.

- -impl_instance_name *impl_inst_name*

  Specifies the hierarchical name of the instance in the RTL.

- -spec_hierarchy_latency *spec_latency*

  Specifies the latency of <*spec_inst_name*>. Default:0.

- -impl_hierarchy_latency *impl_latency*

  Specifies the latency of <*impl_inst_name*>. Default:0.

- -spec_hierarchy_active_cycle *spec_cycle*

  Specifies the cycle number in which <*spec_inst_name*> is scheduled. Default:0.

- -impl_hierarchy_active_cycle *impl_cycle*

  Specifies the cycle number in which <*impl_inst_name*> is scheduled. Default:0.

- -valid_signal *signal_name*

  Specifies the name of the valid signal that will be used to map all the ports.

- -valid_signal_delay *signal_latency*

  Specifies the latency of the valid signal <*signal_name*>.

- -isolate

  Inserts black boxes at signal boundaries of the instances.

- -debug

  Prints informational messages about the different maps created by map_hierarchies.

- **-port_list *tcl_list***

  Specifies the interfaces for hierarchy instances in the specification and implementation design. A few examples are provided below.

Examples

Example 1: Using a common latency value and common valid signal for all interface mappings

Note:
When using a common latency and common active cycle value, the value of the active cycle is used as the latency value for the input port. For output ports, on the other hand, latency is calculated as the sum of latency value of the hierarchy and the value of the active cycle.

Consider the following example:

```
map_hierarchies
 -spec_instance_name top.a
 -impl_instance_name top.b
 -spec_hierarchy_latency 2
 -impl_hierarchy_latency 2
 -spec_hierarchy_active_cycle 1
 -impl_hierarchy_active_cycle 1
 -valid_signal spec.valid1
 -valid_signal_delay 3
 -port_list [list [list in1 in1]
  [list in2 in2] [list out out] [..]]
```

In this example, the specification port top.a.in1 is mapped to the implementation port top.b.in1 with a specification latency of 1 and implementation latency of 1 (see note above for explanation on how latency value was determined); the valid signal spec.valid1 is used with a latency of 3.

Likewise, the input port top.a.in2 is mapped to top.b.in2 with a latency value of 1 and the output port top.a.out is mapped to top.b.out with a latency value of 3 (hierarchy latency plus active cycle value).

In the same example, if the hierarchical names of the specification and implementation instances were not provided, the port list would have to be specified as follows:

```
-port_list [list [list top.a.in1 top.b.in1]
 [list top.a.in2 top.b.in2]
 [list top.a.out top.b.out] [..]]
```

Further, if the map_hierarchies command was passed only with the port list (referencing the full hierarchical name), the ports would have been mapped using the following default values:

- Default latency value of 0.
- Default cycle value of 0.
- No valid signal.

Example 2: Using different latency values and a common valid signal for all interface mappings

Note:
When specific latency values are provided for the ports, SLEC does not calculate the latency. Instead, it uses the latency specified.

Consider the following example:

```
map_hierarchies
 -valid_signal spec.valid1
 -valid_signal_delay 3
 -port_list [list [list top.a.in1 2 top.b.in1 1]
  [list top.a.in2 top.b.in2]
  [list top.a.out top.b.out] [..]]
```

Here, the ports would be mapped using:

- Default cycle value of 0.
- The valid signal spec.valid1 with a latency of 3
- The port top.a.in1 is mapped with a specification latency of 2 and an implementation latency of 1.
- The input ports top.a.in2 and top.b.in2 are mapped using a latency value of 0 (default value of active cycle) while the output ports top.a.out and top.b.out are mapped using the latency value 0 (default hierarchy latency value plus default active cycle value).

Example 3: Using a common latency value and different valid signals for all interface mappings

```
map_hierarchies
 -spec_hierarchy_latency 2
 -impl_hierarchy_latency 1
 -port_list [list [list top.a.in1 top.b.in1 spec.valid1 3 ]
  [list top.a.in2 top.b.in2 spec.valid2 2]
  [list top.a.out top.b.out] [..]]
```

Here, the ports would be mapped using:

- Default cycle value of 0.
- The valid signal spec.valid1 with a latency of 3 is used to map the ports top.a.in1 and top.b.in1.
- The ports top.a.in2, top.b.in2, top.a.out and top.b.out are mapped using the default latency 0 as specific latency values have not been provided.

Example 4: Using different latency and valid signals for all interface mappings

The syntax for this setup is the following:

```
[list [list spec-port1 spec-latency-val1 impl-port1
   impl-latency-val1 valid-signal1 valid-signal-latency1]
  [list spec-port1 spec-latency-val2 impl-port1
   impl-latency-val2 valid-signal2 valid-signal-latency2] [..]]
```

Consider the following example:

```
map_hierarchies
 -port_list [list [list top.a.in1 3 top.b.in1 4 spec.valid1 3 ]
     [list top.a.in2 2 top.b.in2 3 spec.valid2 2]
     [list top.a.out 4 top.b.out spec.valid3 2] [..]]
```

In this example, since the user has provided specific latency values, SLEC does not calculate the latency values. Instead, SLEC assumes the user has taken into account the active cycle value when specifying latency values for the different ports.

map_inputs_with_latency

Maps the specification input port to the implementation input port with the specified latencies.

Usage

map_inputs_with_latency **-spec_latency** ***spec_latency*** **-impl_latency** ***impl_latency***
-unmapped_value ***integer_value spec_input_port impl_input_port***

Arguments

- **-spec_latency** ***spec_latency***

  Specifies the latency for the specification design input port. Default is 0.

- **-impl_latency** ***impl_latency***

  Specifies the latency for the implementation design input port. Default is 0.

- **-unmapped_value** ***integer_value***

  Specifies the value to be applied to the specification and implementation input ports for the unmapped cycles. Default is 0.

- ***spec_input_port***

  Specifies the name of the specification design input port to be mapped.

- ***impl_input_port***

  Specifies the name of the implementation design input port to be mapped.

Description

This command is equivalent to:

```
create_map -input  [create_waveform -sample_start <spec_latency>
 <spec_input_port>] [create_waveform -sample_start <impl_latency>
 <impl_input_port>]
```

with the exception that map_inputs_with_latencies also specifies the value that should be applied to the input ports for the cycles that they are not mapped. If spec_latency or impl_latency are non-zero values, then this command will internally use the create_transactor command to create extra ports at the top level. These ports are then mapped and returned by the map_inputs_with_latencies command so that the user can set additional constraints on these ports, if required.

Consider the following example.

```
map_inputs_with_latency -spec_latency 2 -impl_latency 4
       -unmapped_value 1 spec.input impl.input
```

Here, spec.input with latency 2 is mapped to impl.input with latency 4. Further, the value 1 is applied to both spec.input and impl.input for the cycles that they are not mapped.

The waveforms for this example could look as follows:

```
spec.input: 1  1  s1  s2  s3  s4  s5
impl.input: 1  1  1   1   s1  s2  s3
```

map_with_hold

Maps and constrains the specified specification and implementation design input to the same symbol in any transaction.

Usage

map_with_hold ***spec_input impl_input***

Arguments

- ***spec_input impl_input***

  Inputs from the specification and implementation designs.

Description

In many sequential equivalence checking setups, especially in High-Level Synthesis (HLS) flows, there might be a need to constrain the mapped specification design and implementation design inputs to the same symbol in any given transaction:

- The value of the input needs to be mapped.
- The value needs to be the same symbol during any given transaction.
- The value can change from one transaction to another transaction.

Consider the following example. This example illustrates how a sequence of constraints and maps can be specified more efficiently.

```
map_with_hold spec_input impl_input
```

The above mentioned command is equivalent to specifying the following commands:

```
create_waveform -name w_1 -bitwidth $spec_input_width S+

create_constraint spec.spec_input -waveform [create_waveform -hold_interval
 $spec_throughput_val w_1]

create_constraint impl.impl_input -waveform [create_waveform -hold_interval
 $impl_throughput_val w_1]
```

Where,

- $spec_input is the width of spec.spec_input which is equivalent to the width of impl.impl_input. Note that if the widths of the inputs are not equal, then the map_with_hold command cannot be used.
- $spec_throughput is the throughput of the specification design.
- $impl_throughput is the throughput of the implementation design.

Assuming the throughput of the specification design is 2 and the throughput of the implementation design is 3 and the widths of the inputs being mapped using the map_with_hold command are both 8, then the waveforms could look as follows:

```
spec.spec_input:   <3 3>  <88 88>  <203 203> ...
impl.impl_input:   <3 3 3>  <88 88 88>   <203 203 203> ...
```

Note that there is no way to constrain ports that have been mapped using the map_with_hold command.

mark_memory

Marks memories for symbolic modeling for more efficient verification.

Usage

mark_memory {**-spec** | **-impl**} **-name** ***instance_name*** [-file *source_file*] [-line *line_number*]

Arguments

- **-spec** | **-impl**

  Specifies whether the array is in the specification or implementation design.

- **-name** ***instance_name***

  Specifies the array instance name. For an RTL design, this corresponds with the hierarchical instance name.

- -file *source_file*

  Specifies the source file in which the array is declared.

  Note:
  This argument is only required for C++ designs.

- -line *line_number*

  Specifies the line number in the source file where the array is declared.

  Note:
  This argument is only required for C++ designs.

Description

Large memory arrays in C++ and RTL designs can lead to very long verification times. This is due to explicit modeling of each element in the memory array. To alleviate this problem, you can issue the mark_memory command to mark memory arrays as symbolic. SLEC synthesizes symbolic memories as a single abstract entity, which can dramatically reduce verification runtime. See Modeling Memories for Efficient Verification in the *SLEC User's Manual*.

Examples

Example 1

The following example marks a symbolic memory in an RTL implementation design.

```
mark_memory -impl -name dut.alu_core_inst.RAM_1R1W_inst.mem
```

Example 2

The following example marks a symbolic memory in a C++ specification design.

```
mark_memory -spec -name mem -file fft_rams.cpp -line 23
```

message_filter

Controls the blocking of general SLEC messages from being sent to the selected logs while still allowing certain messages to pass through. Note that warnings and error messages cannot not filtered via this command.

Usage

message_filter -disable -log *log_id* -whitelist *message_id*

Arguments

- -disable

  Turns off filtering imposed by a previous message_filter command. No other options are allowed with this.

- -log *log_id*

  A repeatable item that specifies to which logs to apply the filter. Currently supported values: 'stdout', 'tool'. Can be combined with -whitelist.

- -whitelist *message_id*

  A repeatable item that specifies which messages will not be filtered out. Can be combined with -log.

Description

The typical usage of message_filter is as follows:

```
message_filter -log stdout -whitelist ORC-SPN
... commands that are to have filtered output ...
message_filter -disable
```

This will disable message output to standard output (but allow any instances of the message ORC-SPN) for the group of commands between the two message_filter commands. After that normal messaging is resumed.

print_status

Reports verification status.

Usage

print_status [-summary]

Arguments

- -summary

  Specifies that summarized verification results be printed.

Description

The print_status command reports verification status and results.The example below shows the output of the print_status command when 3 output maps are fully proven. This shows the proof status, name of the mapped points and the latency and throughput for each map in the following format [with format (latency,throughput)]:

```
print_status

[FULL PROOF] Output-pair: spec.Out1 (1,1) impl.Out1 (1,3)
[FULL PROOF] Output-pair: spec.Out2 (1,1) impl.Out2 (1,3)
[FULL PROOF] Output-pair: spec.Out3 (1,1) impl.Out3 (1,3)
```

When used with the -summary option, the print_status command prints the full table of results, as given at the end of verification. A sample summary table is shown below.

```
print_status -summary
Summary of key results:
================================================================================
****************** Design Rule Violations (calypto/ccheck.log) ***************
================================================================================
                   Errors   Warnings
Spec                    0          2
Impl                    0          2
================================================================================
************* Design Characteristics (calypto/characteristics.log) ************
================================================================================
    Inputs/BBox-outputs   Outputs/BBox-Inputs    Inouts    WL-Flops    BL-Flops
Spec                  5                     2         0           2           8
Impl                  5                     2         0           2           8
================================================================================
************************* Mapping (calypto/mapping.log) **********************
================================================================================
     Input-Maps     Output-Maps     Active Flop-Maps    Inactive Flop-Maps
              2               2                    0                     0
Unreachable Input-Maps     Memory-maps
                     0               0
     Unmapped Inputs      Unmapped Outputs      Unmapped Flops
```

```
Spec               3                    0                    2
Impl               3                    0                    2
================================================================================
***************************** Reset (calypto/reset.log) ********************
================================================================================
           Reset WL-Flops      Unreset WL-Flops
Spec                    2                    0
Impl                    2                    0
================================================================================
****************** Equivalence Results (calypto/results.log) ****************
================================================================================
                 Proven  Cond-Proven  Bounded-Proven   Falsified  Not-Attempted
Output-Maps           0            0               0           2              0
Active Flop-Maps      0            0               0           0              0
```

quit

Exits the SLEC command shell and terminates SLEC.

Usage

quit [*exit_code*]

Arguments

- *exit_code*

  Value returned by SLEC at exit. If not specified, it defaults to 0.

Description

The quit command causes SLEC to terminate the session and return control to the operating system command prompt. The exit code is returned during exit. The quit command is an alias for the exit command.

For example, running the following command terminates SLEC and returns an exit code of 27.

```
quit 27
```

On the other hand, running the quit command without an exit code will terminate the SLEC session and return an exit code of 0.

```
quit
```

read_db

Reads and restores a previously built database.

Usage

read_db ***file***

Arguments

- ***file***

  Specifies the name of the previously written database.

Description

The read_db command can be used instead of reading and building specification and implementation designs.

For example, the following command reads the design database *file.db*.

```
read_db file.db
```

Database files can be read using relative or absolute pathnames.

```
read_db ./sub-dir/file.db
```

Using this command can speed up run times. In this example, if the database file exists, it is read. Otherwise the design source code is read and elaborated. The latter usually takes longer than reading the database file. If using this example, remember to delete the database file if the source code was modified otherwise the old database file will be read instead of the new code.

```
set db_name design.db
        if {[file exists $db_name]} {
            # Read the DB file
            read_db $db_name
        } else {
            # else build design and write a new DB file
            build_design -spec -f spec_files
            build_design -impl -f impl_files
            write_db $db_name
        }
```

Note that the read_db command cannot be used along with the read_design or build_design commands.

read_design

Reads in a set of design files into a SLEC design library.

Usage

read_design {**-spec** | **-impl** | **-cons** [-verilog | -vhdl | -systemc | -sv]} ***arguments***...

Arguments

- **-spec** | **-impl** | **-cons**

  Adds modules to the design library specified. Specify -spec to add modules to the specification design library, -impl to add modules to the implementation design library, and -cons to add modules to the constraints design library.

- -verilog | -vhdl | -systemc | -sv

  Specifies the language of the design files being read in. If omitted, the suffix of the first file read is used to establish the design language (Verilog if *.v*, VHDL if *.vhd**, SystemC if *.h** or *.c**, and SystemVerilog if *.sv*).

Verilog and SystemVerilog Arguments

- +define+*macro* [=*macro_body*]

  Defines a macro.

- -f*list_file*

  Specifies the name of the file that lists the set of files and options to be read in.

- -y*dir_name*

  Specifies that SLEC search for unresolved modules in the directory <*dir_name*>.

- -v*file_name*

  Specifies that SLEC search for unresolved modules in the file <*file_name*>.

- +libext+*ext*

  Specifies that SLEC search for unresolved modules in a library directory using extension <*ext*>.

- +incdir+*dir* ...

  Specifies that SLEC search + separated directories to resolve `include directives.

- {-svlog_2005 | -svlog_2009 | -svlog_2012 | -svlog_2017}

  Overrides the default SystemVerilog version set with the system_verilog_version global.

- {-vlog_1995 | -vlog_2001 | -vlog_2005}

  Overrides the default Verilog version set with the verilog_version global.

- *filenames*...

  Specifies a whitespace separated list of design files to be read into SLEC which may include absolute or relative directory paths. By default, any additional file references included within the design files are searched for in the current working directory, and additional include paths may be included using +incdir+<*dir*> options.

VHDL Arguments

- -87

  Specifies that the files should be read in using VHDL 87 standards. Unless specified, the files will be read in using VHDL 93 standards.

- -library *library*

  Specifies that the VHDL units being read in belong to the VHDL logical library <*library*>.

- {-vhdl_1987 | -vhdl_1993 | -vhdl_2000 | -vhdl_2002 | -vhdl_2008}

  Overrides the default VHDL version set with the vhdl_version global.

- *filenames*...

  Specifies a whitespace separated list of design files to be read into SLEC which may include absolute or relative directory paths.

SystemC Arguments

- -I*include-dir*

  Species that SLEC append the specified directory to the include file search path. Do not include SystemC header files (*systemc.h* or the files included by *systemc.h*) because they may interfere with the correct operation of SLEC. SLEC uses its own SystemC header files.

- -std=c++98 | -std=c++11

  Specifies the C++ language version.

  - c++98 : use default C++ version.
  - c++11 : enables support for C++11 constructs.

- -D*macro*, -D*macro*=*value*

  Defines a macro.

- -w

  Specifies that common warnings be suppressed.

- *filenames*...

  Specifies a whitespace separated list of design files to be read into SLEC which may include absolute or relative directory paths. By default, any additional file references included within the design files are searched for in the current working directory, and additional include paths may be included using the -I<*include-dir*> option.

Other options, such as -U<*macro*>, are not understood by the reader.

The include path for *systemc.h* is automatically included and should not be specified.

Description

The read_design command reads in a set of design files. Design modules found within the files are parsed and added to the specified design.

A few points to keep in mind:

- Within a design, all modules must be described in the same language.
- After reading in the design, it must be linked and built using the link_design and build_db commands respectively.
- Before linking a Verilog or VHDL design, multiple read_design commands may be issued. However, all files belonging to a specific library must first be linked and built before linking and building design files of another library. For example, if building a design from the specification library, all files in the specification library must first be built and linked before linking and building files from the implementation library.
- When reading VHDL designs, it is suggested that the read_design, link_design and build_db commands be used instead of the build_design command. This is because the build_design does not support the use of VHDL libraries other than WORK.
- Modules may be referred to within other module definitions.
- If a module contains non-synthesizable code, the module should be blackboxed before it is read in. Otherwise, an error will occur. See the create_black_box command for more information.

Any remaining command arguments are passed as options to the language-specific reader. Options not understood by the reader might be ignored or can lead to an error.

Examples

Example 1: Reads in two Verilog files and places the modules in the implementation design.

```
read_design -impl -verilog transpose_impl.v other_modules_impl.v
```

Example 2: Reads in a SystemC file (inferred by the *.cpp* extension) and places the modules in the specification design. The subdirectories *inc* and *hdr* are appended to the include file search path and the macro constant NODEBUG is defined for this file.

```
read_design -spec -I./inc -I./hdr -DNODEBUG transpose.cpp
```

Example 3: Reads in the SystemVerilog file, *design.v*. Since the design file uses a *.v* extension, the -sv option needs to be explicitly specified.

```
read_design -sv -spec design.v
```

Example 4: Reads a Verilog design into SLEC. After reading the design into the specification library, the design is linked and built.

```
read_design -spec design.v
link_design -spec
build_db -spec
```

Example 5: Reads different VHDL files into different VHDL logical libraries.

```
read_design -impl -library LIB1 entity1.vhdl
read_design -impl -library LIB2 entity2.vhd
read_design -impl top_entity.vhd
link_design -impl
build_db -impl
```

read_reset_image

Reads a reset image for a design from a simulation trace file. This only works for Verilog designs.

Usage

read_reset_image { **-spec** | **-impl** } **-file** ***file*** **-time** ***time*** [-scope *scope*] [-outfile *outfile*] [-verbose]

Arguments

- **-spec** | **-impl**

  Specifies the specification or implementation design to be read.

- **-file** ***file***

  Specifies the name of the file containing the design simulation values. The file can be either a plain VCD file or a gzip compressed VCD file.

- **-time** ***time***

  Specifies the time value used to find the reset image values in the trace file. The time value is interpreted using the timescale found in the VCD file.

- -scope *scope*

  Specifies a hierarchical instance name indicating a scope within the trace file. This scope should correspond to the section of the trace file which corresponds to the top module instance of the specification or implementation design. If no scope is specified, the entire trace file is used.

- -outfile *outfile*

  If specified, this file is created holding the reset image as a set of set_reset_value commands which can be sourced in a problem setup file.

- -verbose

  Prints a list of flops which are in the design but have no value in the reset image.

Description

The read_reset_image reads a reset image for a design from a simulation trace file. Currently, only the VCD file format is supported. Files can be plain or gzip-compressed VCD files. SLEC assumes a file to be gzip-compressed if it uses the *.vcd.gz* suffix.

When using this command, the specification or implementation design must be specified. The read_reset_image command reads the VCD file for reset values for flops within the design. All flops present in the design, except for memories, are expected to have values in the VCD file. A warning is issued if any non-memory flop values cannot be found in the VCD file. The -verbose option can be used to generate a list of non-memory flops which do not have values in the VCD file.

Consider this example illustrating the use of the read_reset_image command. In this example, the VCD file *tb_wave.vcd* is read into SLEC and the register values at time 200 in the scope tb.dut are used to initialize the registers in the specification design.

```
read_reset_image -spec -time 200 -scope tb.dut -file tb_wave.vcd
```

The -time option is used to set the time index within the VCD trace file. The number is scaled by the timescale value defined in the VCD file. For example, if the *spec.vcd* file contains:

```
$timescale
    1ns
$end
```

then the command:

```
read_reset_image -spec -file spec.vcd -time 27
```

will find values at time 27ns in the file *spec.vcd*.

If a signal is transitioning at the time specified, the value will be the final (right-hand) value at that time step.

If the time specified is beyond the last time value in the file, a warning is issued and the values from the last time point are used.

The -scope option can be used to isolate a portion of the VCD file. A hierarchical instance name specifies the portion of the design used. This instance corresponds to the top-level module instance in the specification or implementation design. If no scope is specified, the entire scope is used.

The reset image can be stored in a custom TCL file using the -outfile option. The file contains a sequence of set_reset_value commands to set the flop values within the design. This TCL file can be sourced instead of reading the values directly from the VCD file for each verification run. The -outfile argument *reset_all_dump.tcl* will be recognized in a special way as it is allowing the use of the command without reading in the designs first. That way, all signals with their corresponding values are captured and output.

The read_reset_image command can be used with additional set_reset_value commands. The additional values are set after the reset image has been applied, even if the set_reset_value commands occur before the read_reset_image command in the TCL script.

The read_reset_image command cannot be used in the same verification run with a simulated reset sequence using reset constraints.

Reset images from a VCD file can be converted to a set of set_reset_value commands. These commands are saved to the Tcl file *spec_reset_value.tcl* in this example. Sourcing this generated Tcl file in a subsequent SLEC run can be faster than reading the VCD file. The source command commented out in the example below can then be used in place of the read_reset_image command.

```
read_reset_image -spec -time 200 -scope tb.dut \
        -outfile spec_reset_values.tcl -file tb_wave.vcd

#In the next SLEC run, use this command instead of read_reset_image
#source spec_reset_values.tcl
```

report_globals

Reports global variable settings.

Usage

report_globals [-filename *filename*]

Arguments

- -filename *filename*

  Specifies the name of the file to which global settings are written. By default, settings are written to the standard output. If the file does not exist, a file by this name is created. If a file by this name already exists, the contents of the file are overwritten.

Description

The report_globals command reports the current setting of all global variables.

By default, the output is displayed on the standard outout. To redirect the output to a file, use the -filename option. In the example below, the output is redirected to the file *global_variables.txt*.

```
report_globals -filename global_variables.txt
```

report_hierarchy

Reports design hierarchy information.

Usage

report_hierarchy [-spec | -impl] [-module *module*] [-depth *depth*] [-start_inst *start_inst*] [-inst | -noinst] [-skip_black_boxed_instances] [-list_format]

Arguments

- -spec | -impl

  Restricts reporting to the specification or implementation designs.

- -module *module*

  Reports starting from the specified module. The default is all top-level modules. Not valid with the -start_inst option.

- -depth *depth*

  Shows the hierarchy up to the specified depth. A depth of 0 will show only the top level. The default is to show all levels.

- -start_inst *start_inst*

  Reports starting from the specified instance. Not valid with the -module option.

  If the design has been built, and -spec or -impl has been specified, then the starting instance should be found in the top-level instance of the design. If no design option is specified, the instance should start with either "spec." or "impl.".

  If the design has not been built, -spec or -impl must be specified, and the starting instance can be found in any module in the design.

- -inst | -noinst

  Show (default) or do not show the instance names along with the module names.

- -skip_black_boxed_instances

  Skips reporting of black box instances. By default, these instances are reported. Only relevant with the -inst option.

- -list_format

  Returns the output in list format rather than the default report style.

Description

The report_hierarchy command reports design hierarchy information in report or list format.

Examples

```
slec> report_hierarchy -spec

spec(C)
    b2_c(B)
        a2_b(A)
```

```
        a1_b(A)
    a_c(A)
    b1_c(B)
        a2_b(A)
        a1_b(A)

slec> report_hierarchy -spec -noinst -list_format

C { B { A { } A { } } A { } B { A { } A { } } }

slec> report_hierarchy -spec -noinst -depth 1 -list_format

C { B { } A { } B { } }

slec> report_hierarchy -spec -module B -list_format

B { A { } A { } }
```

report_hierarchy_with_area

Reports the hierarchical blocks in a design with the area for each block.

Usage

report_hierarchy_with_area {**-spec** | **-impl**} [-depth *depth*] [-start_inst *start_inst*]

Arguments

- **-spec** | **-impl**

  Determines whether reporting is to be done for the specification or implementation design.

- -depth *depth*

  Determines the number of levels reported. Setting depth to 0 will show only the top level. By default, all levels are reported.

- -start_inst *start_inst*

  Specifies which instance reporting should start from.

Description

Reports the hierarchical blocks in a design with the area for each block. This allows for easier analysis when looking for large blocks that may cause capacity issues.

Consider the following example of a report generated by the report_hierarchy_with_area command.

```
slec> report_hierarchy_with_area -spec -depth 2
   spec (multi_hier)    1162
      U03 (mm_unit)     156
      U02 (sel_block)   68
      U06 (space_cv)    233
      U05 (mx_unit)     42
      U01 (mm_unit)     156
      U04 (space_cv)    233
      U07 (blender)     267
```

report_messages

Reports the number of times a message has been issued.

Usage

report_messages [-filename *filename*]

Arguments

- -filename *filename*

  Specifies the name of the file the report is to be written to. By default, message statistics are written to the standard output. If a filename is provided, message statistics are written to the file. If the file does not exist, a file by this name is created. If a file by this name already exists, the contents of the file are overwritten.

Description

The report_messages command reports the number of times a message has been issued. A sample report is shown below.

```
slec> report_messages
Message Tally:

APP-CCFRF    = 13
APP-CCP      = 1
APP-CCPS     = 6
APP-FWR      = 1
APP-WDR      = 1
CDB-CCLF     = 4
CDB-CCLH     = 4
CDB-CCS      = 12
VLOG-1005    = 3
VLOG-1007    = 6
VLOG-GVM     = 9
VLOG-PDF     = 3
```

To direct these results to a file, use the -filename option. In the following example, the results are redirected to the *message_file.txt* file.

```
report_message  -filename  message_file.txt
```

report_rule_based_mapping

Provides a report containing the name-mapping rules and the list of flops that will be mapped/unmapped with each rule.

Usage

report_rule_based_mapping [-filename *filename*]

Arguments

- -filename *filename*

  Provides the name of the log file to report the mappings. If not specified, SLEC reports the data into *calypto/rule_based_maps.log*.

Description

The report_rule_based_mapping command processes the namemap rules and provides a report based on that. This command does not populate the final SLEC mapping table used by the formal engine. That is done automatically during verify call. This command can be used in a feedback loop with the create_namemap_rules to ensure that SLEC identifies all the name-based flop maps that the user desires. This command must be invoked after build_design of both the designs. It can be invoked multiple times.

Examples

This command can be used to identify all the flops that will be mapped through name-based mapping, for example:

```
slec> report_rule_based_mapping
```

The path to the log file gets printed on screen.

```
[PSS-RNR]     Reporting all the name mapping-rules and the applicable
 flop-pairs into "calypto/rule_based_maps.log".
```

This user can also provide the name of the report file, for example:

```
slec> report_rule_based_mapping -filename rules.log
```

The following gets displayed.

```
[PSS-RNR]     Reporting all the name mapping-rules and the applicable
 flop-pairs into "rules.log".
```

run_subsession

The TCL command run_subsession is used to run a child process. The command pauses the current SLEC process, starts a child process to run a set of commands, and then returns control to the parent process when the child process has completed.

Usage

run_subsession [-name *name*] [-before {*job* | list *jobs*}] [-after {*job* | list *jobs*}] [-eval *string*] [-program *path*] [-timeout *num*] [-memout *num*] [-pwd *path*]
[-workdir *path*] [-stdout *path*] [-stderr *path*] [-tclscript *path*] [-transfer_globals yes] [-verbose] [-async] ***script.tcl arg1 arg2***

Arguments

- -name *name*

  Specifies the name of the job. If you do not specify this argument, the job name defaults to the handle.

- -before {*job* | list *jobs*}

  The new job must finish before the specified job or list of jobs can begin.

- -after {*job* | list *jobs*}

  The new job can start only after completion of the specified job or list of jobs.

- -eval *string*

  Passes the specified string for evaluation to SLEC at startup. For example:

  ```
  run_subsession -eval "set_global sim_based_validation 0"
  ```

  results in the following call:

  ```
  slec -eval "set_global sim_based_validation 0"
  ```

- -program *path*

  Runs the program executable saved at the specified path. This option can be used only with the -async option.

- -timeout *num*

  Specifies the time limit for the subsession, in seconds.

- -memout *num*

  Specifies the memory limit for the subsession, in megabytes.

- -pwd *path*

  Specifies the path that will be used to set the initial directory when the subsession process is started. If this option is not specified, the initial directory will be set to the workdir of the parent session. If the path is specified as a relative path, the initial working directory of the subsession will be set relative to the workdir of the parent session.

- -workdir *path*

Specifies the path to be used as the subsession's workdir. If not specified, it will default to *<jobname>* within the subsession's initial directory (see the -pwd option). This option only applies when the SLEC executable is the subsession, and so will have no effect if the -program option is specified.

- -stdout *path*

  Specifies the path for the file which captures the subsession's stdout. By default, the stdout is not captured in any file.

- -stderr *path*

  Specifies the path for the file which captures the subsession's stderr. By default, the stderr is not captured in any file.

- -tclscript *path*

  Specifies the path to the TCL script which is to be executed. If specified, *<jobname>* will be inferred from this parameter. Note that this can be omitted and the TCL script can simply be specified on the subsession's command line.

- -transfer_globals yes

  If specified, all globals from the parent process will be transferred into the subsession. This is off by default.

- -verbose

  If you specify this argument, the standard and error output from the subsession is echoed in the parent's output. If you do not specify this argument, the stdout and stderr logs are not generated.

- -async

  Specifies that the subsession is to be run asyncronously with distributed processing. The job is queued and control is returned to the parent immediately. Launch all queued jobs in parallel with the join_async_subsessions command.

The *<jobname>* value is determined by the value specified with the -name argument. If you do not include the -name argument, *<jobname>* is inferred from the name of the TCL script. If multiple jobs are invoked with the same TCL script, SLEC makes a reasonable attempt to uniquify *<jobname>*, if there is a potential for conflict in the subsession's working directory.

Note:
If you do not specify the -async option, child processes share the license of the parent process, and no additional licenses are required. If you do specify the -async option, additional licenses may be required. See Subsession-level Distributed Processing in the *SLEC User's Manual*.

Examples

Example 1

The following example creates a job named "newjob" which must finish before job1 can start.

```
run_subsession -name newjob -before job1
```

Example 2

The following example creates a job called "newjob" which can start only after job1, job2 and job3 have completed.

```
run_subsession -name newjob -after [list job1 job2 job3]
```

Related Topics

Distributed Processing in SLEC [Sequential Logic Equivalence Checker (SLEC) User's Manual]

run_testbench

Invokes third party simulators to validate a SLEC counterexample in the specified trace directory.

Usage

run_testbench [-spec | -impl] [-inc | -mds | -osc | -vcs] [*trace_dir*]

Arguments

- -spec | -impl

  Specifies whether to simulate on the specification or implementation design. If you do not specify this argument, SLEC initiates simulation on both designs.

- -inc | -mds | -osc | -vcs

  Specifies the third-party simulator tool suite to use for testbench simulation: Incisive (inc), Modelsim (mds), OSCI (osc), or VCS (vcs). If you do not specify a simulator, SLEC determines the simulator based on the design language.

- *trace_dir*

  Specifies the counterexample trace directory. SLEC resolves the trace directory as follows:

  - If the current directory is a trace directory and *trace_dir* is not specified, select the current directory.
  - If the current directory is not a trace directory and *trace_dir* is not specified, select the latest trace generated.
  - If *trace_dir* is a relative path, select *./<trace_dir>*.
  - If *trace_dir* does not begin with "trace_" prefix, select *./trace_<trace_dir>*.
  - If *trace_dir* is an absolute path, select *<trace_dir>*.

  See Viewing Waveforms in the *SLEC User's Manual* for a full explanation of how SLEC resolves the trace directory.

Description

The run_testbench command invokes third-party simulators to simulate a counterexample. The SLEC engines, such as simulation-based validation, counterexample replay, concolic engine, and formal engines, can generate multiple counterexamples on a given run. SLEC may also generate multiple counterexamples when the global variable stop_at_first_falsification is 0. Check the contents of *slec.log* to determine trace directories for counterexamples. You can specify the counterexample directory in the *trace_dir* field above.

set_clock_domain

Associates clock edge sensitivity with primary input ports or black box output ports of the design. This command can only be called before a constraint or map command is issued.

Usage

set_clock_domain **-clock *clock*** [-polarity *polarity*] [-spec | -impl] ***port_list***

Arguments

- **-clock *clock***

  Specifies the name of an ideal clock.

- -polarity *polarity*

  Specifies the clock edge sensitivity. A value of 0 indicates the input port waveforms change after the rising edge of clock, while a value of 1 indicates the inputs change after the falling edge. If not specified, the default is rising edge(0).

- -spec | -impl ***port_list***

  Specifies the list of design ports. Alternatively, a single expression using wildcards may be expanded as the list.

Description

Ports are associated with the rising or falling edge of a clock waveform. Only the clock domains of primary input ports and blackboxed output ports may be set by the set_clock_domain command.

If no clocks are defined, it is assumed that all input ports are sensitive to the default clock, _Ideal_Clock_. If only one clock is defined, it is assumed that all input ports are sensitive to that clock.

The clocking network is analyzed to determine the likely clock and polarity of each input. If the input fans out to flops which are synchronized by the positive edge of a clock, then the input is assumed to be synchronized to the same positive edge clock domain. In contrast, if the input fans out to latches which are enabled by the positive edge of a clock, then the input is assumed to be synchronized to the negative edge of the clock.

Setting the clock domain of a primary input or black box output is required when the clock domain cannot be correctly inferred by SLEC, which could happen in either of the following scenarios:

- If there are flops/latches in the design which are sensitive to both phases of the clock.
- If there is clock gating in the clock network.

If -spec or -impl is specified before the port list, all port names in the list are relative to the spec or impl top module instances. For example,

```
set_clock_domain -clock clock -spec { data_in watchdog_enable_in }
```

would refer to the data_in and the watchdog_enable_in ports of the spec instance.

Alternatively, if the -spec or -impl options are omitted, names must include their top module instance name. The previous port list would become:

```
set_clock_domain -clock clock { spec.data_in spec.watchdog_enable_in }
```

Using this form, the list can include ports from both specification and implementation designs. If only one port is to be specified, the list brackets can be omitted.

Consider the following example. In this example, since the data for the specification ports In1 and In3 are associated with the POSITIVE edge of MAIN_CLK, SLEC ensures that the data to the specification ports In1 and In3 are changed at the correct times with respect to MAIN_CLK.

```
create_clock -name MAIN_CLK -period 200
set_clock_domain -clock MAIN_CLK {spec.In1 spec.In3}
```

To associate the data on the implementation port In2 with the NEGATIVE edge of MAIN_CLK, use the -polarity option.

```
create_clock -name MAIN_CLK -period 200
set_clock_domain -clock MAIN_CLK -polarity 1 {impl.In2}
```

set_clock_root

Associates an ideal clock waveform with a primary clock input or blackboxed output port of the design. This command must be specified before any constraint or map commands are used.

Usage

set_clock_root **-clock *clock*** [-polarity *polarity*] [-enable *string*] ***port_list***

Arguments

- **-clock *clock***

  Specifies the name of an ideal clock.

- -polarity *polarity*

  Specifies the clock polarity for the ports specified by <port_list>. Legal values are 0 and 1, where 0 denotes normal/in-phase polarity and 1 denotes inverted polarity. If not specified, polarity is assumed to be 0.

- -enable *string*

  This is an advanced option used to model multiple clocks. This option enables the set_clock_root command to use all the flops and latches fed by the clock port using the list specified by <*port_list*>.

- ***port_list***

  Specifies a list of primary clock input or blackboxed output ports. Alternatively, a single expression using wildcards may be specified.

Description

The set_clock_root command associates an ideal clock waveform, created using the create_clock command, with a primary clock input or blackboxed output ports of the designs. For example:

```
create_clock -name MAIN_CLK -period 200
set_clock_root -clock MAIN_CLK spec.CLK_IN
set_clock_root -clock MAIN_CLK impl.CLK_IN
```

In this example, after the clock MAIN_CLK is defined, the set_clock_root command is used to drive the specification port CLK_IN and the implementation port CLOCK with the clock MAIN_CLK.

Clock ports are associated with the named clock. For example:

```
set_clock_root -clock clock { spec.clk_in spec.timer_clk_in }
```

The list can include clock ports from both the specification and implementation designs. If only one port is specified, the list brackets can be omitted. For example,

```
set_clock_root -clock clock spec.clk_in
```

The polarity option determines whether the ports receive an in-phase (-polarity 0) or inverted (-polarity 1) clock signal. The default polarity is in-phase.

All ports which are determined to be clock input ports must be associated with an ideal clock waveform. Any clock input ports not explicitly associated with an ideal clock must have a clock associated with it using the following rules:

- Unassociated clock ports will be associated with the fastest clocks that are already associated with other clock ports in the design.
- If no clocks are associated with the design, the default ideal clock will be associated with all clock ports in the design.

All input and black box output ports which are determined to be clock input ports must be associated with an ideal clock waveform. If these ports are not explicitly associated with a clock waveform using the set_clock_root command, the default clock waveform will be associated with them.

Modeling Multiple Clocks

Only one clock waveform can be created in SLEC. If the consequence of two or more clock ports being fed by different clock waveforms needs to be functionally verified, then the -enable Consider the following example. If an implementation design has two clock ports clk1 and clk2 which feed flops f1 and f2 respectively. Then, the following set of commands will achieve the effect of driving the flops driven directly or indirectly by clock ports with uncorrelated clock waveforms:

```
set_clock_root -clock CLK -enable EN1 impl.clk1
set_clock_root -clock CLK -enable EN2 impl.clk2
```

When the specified enable string is new, SLEC creates a new input port which is used as an enable condition for all of the flops fed by the clock. So flop impl.f1 will be enabled when the input port associated with EN1 is high and likewise impl.f2 will be enabled when the input port associated with EN2 is high.

Multiple clocks can be driven by the same enable condition. For instance, in the above example, if there is a third clock port impl.clk3 and it needs to have the same clock waveform as impl.clk1, the following command can be specified:

```
set_clock_root -clock CLK -enable EN1 impl.clk3
```

The enable conditions can be reused across the specification and implementation designs. A useful way to think about the enable condition is that the set_clock_root command without the -enable option is always enabled.

set_constant

Forces a signal or a slice of it, in a design module or instance, to a specified constant or symbolic value.

Usage

set_constant { { **-spec** | **-impl** } **-module** ***module*** | **-inst** ***inst*** } **-value** ***value*** [-ignore_in_testbench] ***signal***

Arguments

- **-spec** | **-impl**

  Specifies whether the module exists in the specification library or the implementation library. This option requires the -module option to be specified.

- **-module** ***module***

  Specifies the module containing the signal to be set constant. This option requires the -spec or -impl option to be specified. Note that the -module and -inst options are mutually exclusive and cannot be used together.

- **-inst** ***inst***

  Specifies the instance containing the signal to be set constant. Note that the -inst and -module options are mutually exclusive and cannot be used together.

- **-value** ***value***

  Specifies a valid Verilog constant or a symbolic value created using the create_symbolic_value command. If an invalid constant or nonexistent symbolic value is provided, an error is issued.

- -ignore_in_testbench

  If set, will ignore forcing the signal with the constant value in testbench.

- ***signal***

  Specifies the port or signal to force to a constant value. Verilog designs can set any port, wire, or reg while SystemC designs can set any port or sc_signal. If the signal expression represents a slice or a bit of a signal, then those bits of the signal are set to the constant. The syntax for expressing the slices follows that of the language of the module in which the signal exists. For example, if setting a constant on a signal, out_sig, in a Verilog module, the user will need to specify out_sig[1] or out_sig[5:3]. If out_sig was on a VHDL module, the user would specify out_sig(1), out_sig(5 downto 3).

Description

The set_constant command sets a port in a specified module/instance to a constant value. In case of a module, all instances of the specified module are affected. In case of an instance, only the specified instance is affected. If the value is a symbolic-value, then set_constant can only be used to constrain primary-inputs.

Here are some examples of the value that can be passed with the -value option. The value must be Verilog constant or a symbolic-value.

27	signed value of at least 7 bits (27)

-27	signed value of at least 7 bits (-27)
4'b0001	4 bit unsigned value (1)
4'hF	4 bit unsigned value (15)
4'sHf	4 bit signed value (-1)
'o37	unsigned value of at least 6 bits (31)
op_sym1	Where op_sym1 was created using: `create_symbolic_value -name op_sym1 -bitwidth 16 -signed`

The following are examples of Verilog constants that are not allowed:

ABCDEF	Invalid decimal value
4'bWXYZ	Invalid binary value

As an example, an up-down counter module updn_counter in the specification design can be forced to only count up by setting its up input port to 1:

```
set_constant -spec -module updn_counter -value 1 up
```

Another example, a bus mybus in a VHDL module bus_if in the specification design can be constrained to have the top 3 bits to be zero, by the following command:

```
set_constant -impl -module bus_if -value 3'b0 mybus(15 downto 13)
```

In the following example, the output-port opcode of a black box instance spec.bb1 is forced to a constant 9.

```
set_constant -inst spec.bb1 -value 9 opcode
```

In the following example, input-ports op1 in both the specification designs are constrained to be the same symbolic-value for the entire verification, while the input-port op2 in the specification design is constrained to be a different symbol.

```
create_symbolic_value -bitwidth 4 -name op1_sym

set_constant -spec -module alutop -value op1_sym op1
set_constant -impl -module alutop -value op1_sym op1

create_symbolic_value -bitwidth 4 -name op2_sym
set_constant -spec -module alutop -value op2_sym op2
```

In the following example, the set_constant command is used to assign the constant value 0 to the port busy of the module fir_filter of the specification design.

```
set_constant -value 'h0 -spec -module fir_filter  busy
```

Consider another example where the -inst option is used to set a constant value of 4 on the input port I2 of the implementation instance U1.

```
set_constant -value 4 -inst impl.U1 I2
```

set_data_type

Sets the data type for the specified port or signal.

Usage

set_data_type ***signal type***

Arguments

- ***signal***

  Specifies the port or signal.

- ***type***

  Specifies the data type. Valid values are:

  - -float: IEEE 16/32/64-bit half/single/double precision
  - -bfloat: 16-bit "brain" float
  - -int: 8/16/32/64-bit signed integer
  - -uint: 8/16/32/64-bit unsigned integer

Description

The set_data_type command sets the designated port, signal, or slice to the specified data type. Currently, SLEC uses the data type information on primary inputs to simulate various corner cases for given data types during simulation based validation. In the future, the usage of type information in the verification flow may be expanded.

Examples

Example 1

The following example sets all 64 bits of signal 'i' to type float:

```
set_data_type -float spec.i
```

Example 2

The following example sets only a slice of the signal to type float:

```
set_data_type -float spec.i[35:20]
```

set_design_scope

Sets the design scope for the top module. This is useful when the top module requires additional information from its containing hierarchy.

Usage

set_design_scope **-spec *design_scope*** | **-impl *design_scope***

Arguments

- **-spec *design_scope*** | **-impl *design_scope***

  Specifies the name of the containing hierarchy that has the additional context information required by the top module. Specify -spec to limit the scope to the specification design scope and -impl to limit the scope to the implementation design.

Description

The set_design_scope command sets the design scope for the top module. This is useful when the top module requires additional context information from its containing hierarchy. Consider the following example of a specification design.

```
interface interface1 ( parameter P1 =2);
.....
.....
endinterface

module test;
  interface1 #(.P1( 3)) i1 (clk);
  bot1 #(.X(6)) inst1( i1, clk);
  bot2 #(.Y(7)) inst2( clk );
endmodule
```

In this example, the parameter P1 in the interface interface1 has a default value of 2. Assume that the top module is test. In this scenario, if the module bot1 was to be set as the top module using the "build_design -top" command, i1 would be passed with its default parameter value 2 and not the value 3 as defined within the module test.

The set_design_scope command enables the user to pass the parameter value 3 for the interface port i1, as follows:

```
set_design_scope -spec test
build_design -spec -top bot1 design.v
```

Running these commands will set bot1 as the top module with interface port values from the module test.

Usage Notes

- The set_design_scope command must be issued before linking the design.
- The -defparam option, used while linking a design, cannot be used with the set_design_scope command.
- Multiple instances of the top module in the design scope hierarchy are not supported. Consider the following specification module definition:

```
module test;
  interface1 #(.P1( 3)) i1 (clk);
  bot1 #(.X(6)) inst1( i1, clk);
  bot1 #(.X(8)) inst2( i1, clk);
  bot2 #(.Y(7)) inst3( clk );
endmodule
```

- Setting module bot1 as the top module for the design scope test, as follows, will lead to an error.

```
set_design_scope -spec test
build_design -spec -top bot1 design.v
```

set_global

Sets a global variable value.

Usage

set_global ***name value***

Arguments

- ***name***

  Specifies the name of the global whose value is to be set.

- ***value***

  Specifies the value to be assigned to the global variable.

Description

Global variables are set by the user to alter the default behavior of SLEC. The set_global command sets a new value for a global variable and then returns the variable's value. For globals that accept boolean values, true or 1 enable the global while false or 0 disable the global.

Consider the following example. The following command sets the value of the global variable spec_throughput to 3.

```
set_global spec_throughput 3
```

Refer to “Global Variable Reference” on page 195 for details of currently supported global variables.

set_inductive_state

Indicates that the specified flop is inductive so the flop is guaranteed to go back to its reset value at transaction boundaries.

Usage

set_inductive_state ***flop_name*** [-assume_only]

Arguments

- ***flop_name***

  Specifies the name of the flop which is guaranteed to be inductive.

- -assume_only

  Unconditionally assumes that the specified flop is inductive.

  When this option is specified, in case the designs are proven equivalent, SLEC issues a conditional proof instead of a full proof. This is because the designs are proven equivalent under the assumption that the flops specified by <*flop_name*> are indeed inductive.

Description

For designs with throughputs greater than 1, it is possible that there are flops that have the same constant value at transaction boundaries. This typically happens for flops which are counters and go through a fixed state-transition pattern every transaction. To elaborate, consider the example of a design that has a throughput of 4 and a flop named sched_reg that has a reset-value of 0 and transitions from 0 -> 1 -> 2 -> 3 -> 0 -> 1 -> 2 -> 3 -> 0 ... regardless of the inputs. Because the flop has a fixed transition pattern, it is considered an inductive flop.

For a flop to be deemed inductive, it is imperative that the value for all transactions remain same. In the above example, had the transition been, 0 -> 1 -> 2 -> 3 -> 4 -> 1 -> 2 -> 3 -> 4 ..., the flop would not be deemed inductive because the value in the first cycle of transactions 2 and 3 is 4 which is different from the reset value of 0.

SLEC uses information about the inductive flops to perform optimizations during sequential analysis, making it important to specify the existence of inductive flops in a multi-throughput design. In the state check phase of sequential analysis, SLEC also automatically attempts to infer and formally prove inductive flops. However, as this process is expensive, having prior knowledge of inductive flops reduces the scope of the state checks.

For all flops which have been specified to be inductive, SLEC guarantees that the flops are indeed inductive. In case the flops are found to be non-inductive, a counterexample is generated.

set_reset_length

Explicitly sets the length of the reset sequence for the specification or implementation design.

Usage

set_reset_length { **-spec** | **-impl** } { **-length** ***length*** } [-clock *clock*]

Arguments

- **-spec** | **-impl**

  Specifies whether the reset length is to be applied to the specification or implementation design.

- **-length** ***length***

  Specifies the number of clock cycles for reset.

- -clock *clock*

  Specifies the clock used to count the reset cycles. If the design has a single clock, this option need not be specified.

Description

The set_reset_length command establishes the length of the reset sequence for a design. Reset constraints are applied during the reset sequence of a design. The length of the reset sequence must either be explicitly specified or be inferable from the reset constraints.

To specify an explicit reset length, use "-length <*length*>". For example, the following command sets the reset length for the specification design to 2 cycles of bclk.

```
create_clock -name bclk -period 200
set_reset_length -spec -length 2 -clock bclk
```

If no length is specified, it is inferred from the reset constraints. If all specifications of reset constraints define the constraining waveform for same number of finite cycles then this number is used as the reset length.

For example, these commands infer a reset length of 3 cycles of the single clocked specification design. If the number of finite values differs, then an explicit reset length must be provided. For reset length calculations, the number of finite values is counted as the number of explicit symbols in the waveform discounting a trailing +.

```
create_waveform -name rw_in1 -bitwidth 1 { 1 S 0 }
create_constraint -reset spec.in1 rw_in1
create_waveform -name rw_in2 -bitwidth 1 { S 1 1 }
create_constraint -reset spec.in2 rw_in2
```

So, the following waveforms all have a length of 5:

```
{ 1 0 1 0 1 }
{ 1 S S S 1 }
{ 1 S S S S }
{ 1 S S S 1+ }
{ S S S S S+ }
```

Inputs Under Reset

During the reset sequence, unconstrained or symbolically constrained input ports will normally have X values applied each cycle so as not to over constrain the reset state. This default can be overridden by using the global value_on_input_pins_during_reset. Legal values are 0, 1, x, or r. An r value will randomly apply a 0 or 1 value each cycle. Consider a specification design with in1 and in2 input ports:

```
set_global value_on_input_pins_during_reset 0
create_waveform -name rw_in1 -bitwidth 1 { 1 S 0 }
create_constraint -reset spec.in1 rw_in1
```

The global reset value is set to 0, so all unconstrained and symbolically constrained input values will be set to 0 values. Since port in2 is unconstrained, 0 values will be applied to in2 during the reset sequence. Port in1 is constrained with a symbolic constraint in the second cycle which will be replaced with a 0 value. So, the reset sequence will be:

```
in1: 1 0 0
in2: 0 0 0
```

set_reset_value

Sets flops, latches, and memory locations to their reset values.

Usage

set_reset_value { **-spec** | **-impl** } { **-all** | **-allX** | **-list** ***flops*** } { **-value** ***value*** }

Arguments

- **-spec** | **-impl**

  Specifies whether the command is to be applied to the specification design library or the implementation design library.

- **-all**

  Specifies that the command be applied to all flops, latches, and memory locations in the design.

- **-allX**

  Specifies that the command be applied to all uninitialized flops, latches, and memory locations in the design.

- **-list** ***flops***

  A TCL list of flops, latches, and memory locations specified by their hierarchical instance names. Alternatively, a single expression using wildcards may be expanded as the list.

- **-value** ***value***

  Specifies the reset value.

Description

The set_reset_value command can be used to explicitly set reset values on flops, latches and memory arrays. The following command assigns all uninitialized state elements within the specification design to the value 0.

```
set_reset_value -spec -allX -value 0
```

Assigning values to state elements using the -list option has a higher priority over -all and -allX options. This example will first set all flops and memory arrays in the implementation design to 0, then set the state_vector and ccode registers to a value of 1 and finally set all entries of the lu_mem memory array to a value of -1.

```
set_reset_value -impl -all -value 0
set_reset_value -impl -list {mod1.state_vector mod1.ccode} -value 1
set_reset_value -impl -list {mod2.lu_mem} -value { -1 }
```

set_symbolic_constant

Specifies that SLEC perform the equivalence check with the specified input ports/black box output ports held constant, where the constant value is given by the list or range of values provided.

Usage

set_symbolic_constant **-inputs *port_names*** [-values *Tcl_list_of_values* | *range_of_values*]

Arguments

- **-inputs *port_names***

  Specifies the names of the primary input ports or black box output ports which will be constrained to a symbolic constant. All specified ports should have the same width and sign. There should be atleast one port from the specification design and one port from the implementation design. Sliced ports are not supported.

- -values *Tcl_list_of_values* | *range_of_values*

  Specifies the values or range of values to which the specified port can be constrained. For example, to specify the values 1, 3, or 6 for a 3-bit port, use: -values {1 3 6}.

  Range is provided in the following format :{<*minimum legal value*> - <*maximum legal value*>}.

  For example, to constrain a port to a value between 4 and 62 for a 6-bit port, use: "-values {4 - 62}".

  If a value is not specified, the port will be held constant to any of the values as determined by its datatype.

Description

The set_symbolic_constant command instructs SLEC to do the following:

1. Create a symbol that can assume any of the values in the specified list or a value which belongs to the specified value range.

2. Verify design equivalence with the specified ports constrained to the symbol created.

Consider the following example.

```
set_symbolic_constant -inputs {spec.opcode impl.opcode} -values {3 6}
```

In this example, SLEC verifies the equivalence of the specification and implementation designs with the pair of ports spec.opcode and impl.opcode either constrained to the value 3 forever or constrained to the value 6 forever. This is a convenient alternative to manually performing two runs with spec.opcode and impl.opcode constrained using the set_constant command to 3 and 6, respectively.

Restrictions

- The user must provide at least one specification design and one implementation design port.

- All the ports to be constrained must have the same datatype, that is, the same width and sign.

- The ports listed should not already be constrained using:
  - a prior or subsequent set_constant, set_symbolic_constant, or create_constraint

    OR
  - a map from the "create_map -input" command.
- Sliced ports are not supported.

Examples

Consider the following examples of the set_symbolic_constant command.

Example 1 : Providing specific values

```
set_symbolic_constant -values {1 2} -inputs {spec.in1 impl.in1}
```

In this example, SLEC checks for design equivalence by applying a symbol that could assume the values 1 or 2. The input ports spec.in1 and impl.in1 are constrained using this symbol and the value of the symbol remains fixed through the verification run.

Running this command configures SLEC for a single verification run that is symbolically equivalent to two verification runs with the set_constant command as follows.

Run 1

```
set_constant -spec -module top -value 1 in1
set_constant -impl -module top -value 1 in1
```

Run 2

```
set_constant -spec -module top -value 2 in1
set_constant -impl -module top -value 2 in1
```

Example 2 : Using a range to specify values

```
set_symbolic_constant -values {3 - 7} -inputs {spec.in1 impl.in1}
```

In this example, SLEC checks for design equivalence by applying a symbol that could assume the values 3, 4, 5, 6, or 7. The input ports spec.in1 and impl.in1 are constrained using this symbol and remains fixed through the verification run.

Example 3 : Not specifying any value

```
set_symbolic_constant -inputs {spec.in1 impl.in1}
```

In this example, only the ports are specified. When values are not provided to the set_symbolic_constant command, SLEC verifies design equivalence by creating a symbol whose bit-width and sign is the same as the ports specified. An error is issued if the specification and implementation ports are of unequal widths or different signs.

Example 4 : Running set_symbolic_constant with different sets of ports

```
set_symbolic_constant -values {1 2} -inputs {spec.in1 impl.in1}
set_symbolic_constant -values {3 4} -inputs {spec.in2 impl.in2}
```

In this example, SLEC will verify design equivalence of the specification and implementation designs with the pair of ports spec.in1 and impl.in1 either constrained to the value 1 forever or constrained to the value 2 forever; the pair of ports spec.in2 and impl.in2 will be constrained to the value 3 forever or constrained to the value 4 forever.

Verification in the presence of two symbolic constants is equivalent to doing multiple runs with the cross-product of the set_constant command.

Example 5 : Invalid Usage: Referencing a slice of a port

```
set_symbolic_constant -inputs {{spec.arr[1]} {impl.arr[1]}}
```

SLEC will issue an error as the set_symbolic_constant command does not support sliced ports.

Example 6 : Invalid Usage: Using wildcards in the port name

```
set_symbolic_constant -values {3 7} -inputs {spec.in* impl.in*}
```

SLEC will issue an error as the set_symbolic_constant command does not support the use of wildcards in the port name.

set_verbosity

Sets the amount of information reported by SLEC for each message.

Usage

set_verbosity { **-error** ***message_id*** | ***message_id level*** | ***level*** }

Arguments

- **-error** ***message_id***

  Upgrades the severity of a Warning to an Error.

- ***message_id level***

  A value between 0 and 9. This value overrides the default verbosity level of a message and is used to determine whether or not a message will be displayed.

- ***level***

  A value between 0 and 9 determining whether or not a message will be displayed. For example, if an error message has a verbosity level of 4 and this value is set to 3, the message will not be displayed. On the other hand, if this value is set to 6, this message will be displayed.

Description

This command controls the amount of information reported by SLEC.

Every message in SLEC has a default severity level. Specify -error to upgrade a warning message to an error message.

For example, if SLEC encounters an unsupported Verilog primitive when reading a design, it generates the VLOG-UDE warning message. To change the severity of this message to an error, specify the set_verbosity command as follows:

```
set_verbosity -error VLOG-UDE
```

Each message in SLEC has a default verbosity level and is displayed only if the verbosity level set using the "set_verbosity <*level*>" command is set to that number or lower. For example, the following command sets the verbosity level to 5. All messages with a default verbosity level of 5 and below are displayed.

```
set_verbosity 5
```

The verbosity level of a message can also be changed. For example, the following message will set the verbosity level of this message to 8. This message will be displayed only if the set_verbosity command sets the verbosity level to 8 or lower.

```
set_verbosity SLEC-RDF 8
```

set_verification_mode

Optimizes the verification performance of a flow by setting the required globals.

Usage

set_verification_mode { **-system_synthesis** | **-system_manual** | **-rtl_operator** | **-rtl_bit** }
[-property_checks]

Arguments

- **-system_synthesis** | **-system_manual** | **-rtl_operator** | **-rtl_bit**

  Specify -system_synthesis to optimize verification performance for a pair of designs where the implementation design has been generated from the specification design using a behavioral synthesis tool.

  Specify -system_manual to optimize verification performance for a pair of designs where the specification design is at the system-level and the implementation design has been generated by performing manual transforms on the specification design.

  Specify -rtl_operator to optimize verification performance for a pair of RTL designs at the operator level.

  Note:
  The gate-count to node-count ratio, as reported in *characteristics.log*, for a design at the operator level will be high.

  Specify -rtl_bit to optimize verification performance for a pair of RTL designs where at least one design is a bit-level design.

  Note:
  The gate-count to node-count ratio, as reported in *characteristics.log*, for a design at the bit-level will be close to 1.

- -property_checks

  Instructs SLEC to set the appropriate globals required to verify a design that includes property checks. Note that this option can only be specified with the -system_synthesis and -system_manual options.

Description

The set_verification_mode command sets the globals required to optimize the verification performance for the specified flow. As this command changes global settings to optimize the flow, it is recommended that all set_global commands be specified after the set_verification command.

shutdown_async_worker

Terminates the worker that has the specified handle.

Usage

shutdown_async_worker **-handle *worker_handle***

Arguments

- **-handle *worker_handle***

  Specifies the handle of the worker process.

Description

The command returns void.

If the specified worker handle is valid, the termination is handled silently; otherwise, the details are noted in the *.exec.log* file.

This API can be used from the NotifyIdleCB API in the site config to release the free grid-system resources. If the API is run for a busy worker, the worker must kill its job, place it on the queue, and then terminate.

slice_flop

Slices the named flop along the specified slices to enable subsequent problem setup commands to refer to those flop slices.

Usage

slice_flop ***flop_name*** [*list_of_slices*]

Arguments

- ***flop_name***

  Specifies the name of the flop to be sliced.

- *list_of_slices*

  Specifies the slices along which the flop should be sliced. If not specified, the flop is completely bit-blasted.

Description

The slice_flop command slices a flop in accordance with the specified slices, so that subsequent commands can explicitly refer to the slices of the flops for the purposes of creating flop-maps on, set reset-values of flop slices etc. If no slices are specified, the flop is completely bit-blasted.

Verilog/VHDL style ranges can be specified regardless of the actual design language, but the MSB/LSB consistency is enforced. For example, a big-endian range cannot be specified for a little-endian flop. Relative ordering of the ranges in the list is not required. The range list may be incomplete. For example, only 7:4 can be specified for a flop which is declared as 7:0. In such cases, SLEC auto-infers the rest of the gap(s) and slices them accordingly.

It is legal to slice the same flop multiple times as long as two already-sliced ranges are not resliced. The ranges must not overlap. If there is only 1 range specified and its bounds match the flop being sliced, then no slicing is performed. If the flop has already been specified in a flop map, it cannot be sliced.

The following example shows how the slice_flop command is used to create the slices 1:0, 3:2 and 7:4 of spec.cntrl_data_bus. Subsequent commands to reset and map these slices are also shown.

```
slice_flop spec.cntrl_data_bus 7:4 3:2 1:0

create_map -flop spec.cntrl_data_bus[7:4] impl.cntrl_bus
create_map -flop spec.cntrl_data_bus[3:2] impl.op1
create_map -flop spec.cntrl_data_bus[1:0] impl.op2

set_reset_value -spec -list spec.cntrl_data_bus[7:4] -value X
set_reset_value -impl -list impl.cntrl_bus -value X
The following example illustrates VHDL style slicing:
slice_flop spec.i1.state_val 15 downto 11  10 downto 1   0 downto 0
```

Consider another example where a 3-bit flop in the specification design is to be mapped to three 1-bit flops in the implementation design. This example shows how the flop state in the specification design is sliced into individual bits and then mapped using the create_map command to 1-bit flops in the implementation design.

```
slice_flop spec.state
create_map -flop spec.state[2:2] impl.DUT.proc.state_2
create_map -flop spec.state[1:1] impl.DUT.proc.state_1
create_map -flop spec.state[0:0] impl.DUT.proc.state_0
```

source

Reads commands from the specified file and runs them in the Tcl shell.

Usage

source [-echo | -quiet | -verbose] ***file***

Arguments

- -echo

  Displays the sourced content while executing the commands. The content is displayed on the screen as comments. Results are also written to *slec.log* and the *calypto/slec.cmd* file as comments.

- -quiet

  Suppresses any and all recording of the sourced content to log files.

- -verbose

  Same as -echo.

- ***file***

  The file containing commands to be run.

Description

The source command reads commands from the specified file and executes them in the Tcl shell. It is an enhanced version of the standard Tcl source command. If none of the optional arguments are present, the contents of the sourced file are not echoed to the screen or the log file, but they are written to the *slec.cmd* file as comments.

summarize_slec_status

Displays a snapshot of the current SLEC status.

Usage

summarize_slec_status [-command_line] [-exec_state] [-cpu_time] [-wall_time] [-curr_mem] [-peak_mem] [-constants] [-constraints] [-verify_mode] [-result] [-last_solver] [-cache_memory] [-mapped_signals] [-unmapped_port] [-port_status] [-covers] [-cover_status] [-asc_assumes] [-sva_assumes]

Arguments

- -command_line

  Displays the SLEC command-line as invoked by the user

- -exec_state

  At any point in time, a SLEC run can be in one of the following states:

Table 2. SLEC States

1	INIT
2	BUILD_DESIGN_SPEC
3	BUILD_DESIGN_IMPL
4	BUILD_DESIGN_DONE
5	SETUP
6	VERIFY
7	VERIFY_DONE
8	VERIFY_CASE_SPLIT (parent process for external case splits)

- -cpu_time

  CPU time elapsed in seconds since the start of SLEC run.

- -wall_time

  Wall clock time elapsed in seconds since the start of SLEC run.

- -curr_mem

  Current memory (MB) consumed by SLEC and its processes.

- -peak_mem

  Peak memory (MB) consumed by SLEC and its processes.

- -constants

  Any design inputs set constant by the user

- -constraints

Any design inputs constrained with create_constraint

- -verify_mode

  Verification mode as specified by the user. It currently supports three values:

  FULL_PROOF_MODE, BOUNDED_PROOF_MODE, and BUG_HUNT_MODE.

- -result

  Verification result of the current SLEC run. This attribute encodes one of two values: PASS (if all proof obligations are successfully met) and FAIL.

- -last_solver

  The last verification engine run before the verify command terminated. This attribute can take on any of the following values:

  - 1TU
  - VERIFY_INDUCT_INIT
  - SIMULATION
  - VERIFY_L0
  - VERIFY_LL1
  - VERIFY_LL3
  - VERIFY_HL1
  - VERIFY_HL2
  - VERIFY_HL3
  - VERIFY_BEC
  - VERIFY_CONCOLIC

- -cache_memory

  Solver cache statistics, including:

  - Hit Count
  - Miss Count
  - Saved Runtime (sec)

- -mapped_signal

  Displays the mappings of Input and Output signals in the corresponding spec and impl files

- -unmapped_port

Displays the ports which have no mapping in the corresponding spec and impl files

- -port_status

  Displays the current status of the ports at any point in a SLEC run. The attribute can take any of the following values:

  - PROVEN
  - PROVEN_CONDITIONALLY
  - PROVEN_CONDITIONALLY -1
  - BOUNDED_PROVEN <proof_depth>
  - FALSIFIED <proof_depth >
  - UNKNOWN
  - UNDECIDED
- -covers

  Lists the names of all the cover maps.
- -cover_status

  Displays all the user specified cover_maps along with their status. The statuses could take the following values:

  - UNCOVERABLE
  - NOT_COVERED_UPTO <*proof_depth*>
  - COVERED_AT <*proof_depth*>
  - UNCOVERABLE
- -asc_assumes

  Lists all the assumed inline asserts in C++.
- -sva_assumes

  Lists all the SVA assumptions in RTL

Description

The *summarize_slec_status* command can be invoked at any point in the SLEC flow with or without any flag. The flags are mutually exclusive, i.e., we can provide at most one of them with a *summarize_slec_status* command. When the command is invoked without any flag, all the possible attributes will be displayed. The output of the command is organized as a Tcl array of "{attribute} {value}" pairs that describe the SLEC run. An "{attribute} {value}" pair can be hierarchical where the {value} portion could comprise a list of additional "{attribute} {value}" pairs

The summarize_slec_status output displays the following information:

Examples

Example 1: Results after Falsification

The following example shows the command output after SLEC has falsified one output map.

```
slec> summarize_slec_status
{Command} {test1.tcl } {STATE} {VERIFY_DONE} {CPU} {1} {WallClock} {6}
{Current Memory} {17} {Peak Memory} {60}  {Constants} {  {spec.in} {1'b1}
}  {Constraints} {  } {Mode} {BEC_MODE} {Result} {FAIL} {MappedSignals}
 { {InputMaps} { { { spec.reset } { impl.reset  } } } {OutputMaps}
 { { { spec.out } { impl.out  } }  } } {CoverStatus} {  } {PortStatus}
 { { {FALSIFIED 2} {spec.out} {impl.out} }  } {UnmappedPorts} { {spec.clk}
 {spec.a} {spec.b} {impl.clk} } {AscAssumes} { } {SvaAssumes} { } {Last
 Solver}
{SIMULATION} {Cache Summary} { {Hit Count} {0} {Miss Count} {0} {Saved
 Runtime} {0} }
```

Example 2: Parsing Output

The output of the *summarize_slec_status* is organized as a Tcl array. The following example shows how to parse the array.

```
slec> set summary [summarize_slec_status]
# To obtain an individual value of an attribute, query as follows:
slec> array set val $summary
slec> foreach index [array names val] {
....     puts "val($index): $val($index)"
....  }
```

Executing this code after obtaining the results in Example 1 generates the following output:

```
val(Constraints):
val(STATE): VERIFY_DONE
val(Constants):   {spec.in} {1'b1}
val(Result): FAIL
val(MappedSignals): {InputMaps} { { { spec.reset } { impl.reset  } } }
 {OutputMaps} { { { spec.out } { impl.out  } }  }
val(CoverStatus):
val(PortStatus): { {FALSIFIED 2} {spec.out} {impl.out} }
val(UnmappedPorts): {spec.clk} {spec.a} {spec.b} {impl.clk}
val(AscAssumes):
val(SvaAssumes):
val(Last Solver): SIMULATION
val(CPU): 1
val(Peak Memory): 60
val(Cache Summary):  {Hit Count} {0} {Miss Count} {0} {Saved Runtime} {0}
val(WallClock): 20
val(Current Memory): 17
val(Mode): BEC_MODE
val(Command): test1.tcl
```

Example 3: Reading Results from Tcl Script

The following example demonstrates several Tcl methods of accessing the results when the key contains a space. It also shows several incorrect methods to avoid.

```
slec> set summary [summarize_slec_status]
# To obtain an individual value of an attribute, query the results in one
 of the following Tcl formats when the associative array key contains a
 space:

slec> array set val $summary

# Method 1
info exists val(Last\ Solver)

# Method 2
info exists "val(Last Solver)"

# Method 3
info exists {val(Last Solver)}

# Method 4
set ls "Last Solver"
info exists val($ls)

# The following are INCORRECT methods of accessing the array values:
# Method 5 : Bad Tcl. Error.
info exists val(Last Solver)

# Method 6 : Bad Tcl. Error.
info exists val("Last Solver")

# Method 7 : Bad Tcl. Error.
info exists val({Last Solver})
```

Example 4: Querying Constants

All occurrences of the SLEC command *set_constant* are captured in the named attribute "Constants". Query all constants in a SLEC run as follows:

```
slec> array set const $val(Constants)
slec> foreach index [array names const] {
....     puts "const($index) : $const($index)"
....  }
const(spec.in) : 1'b1
```

Example 5: Querying Constraint Waveforms

All input variables constrained via waveforms are captured under the Constraint attribute. Query all input constraints as follows:

```
slec> array set cons $val(Constraints)
slec> foreach index [array names cons] {
....    puts "cons($index) : $cons($index)"
....  }
cons(specwf_a) :  {specwf} {spec.a}
cons(a1_impl_reset) :  {a1} {impl.reset}
cons(a1_spec_reset) :  {a1} {spec.reset}
cons(a2_impl_reset) :  {a2} {impl.reset}
cons(a2_spec_reset) :  {a2} {spec.reset}
```

Alternatively, the constraint waveforms could be captured by the -constraints flag in the following manner:

```
slec> array set cons [summarize_slec_status -constraints]
slec> foreach index [array names cons] {
....    puts "cons($index) : $cons($index)"
....  }
cons(specwf_a) :  {specwf} {spec.a}
cons(a1_impl_reset) :  {a1} {impl.reset}
cons(a1_spec_reset) :  {a1} {spec.reset}
cons(a2_impl_reset) :  {a2} {impl.reset}
cons(a2_spec_reset) :  {a2} {spec.reset}
```

unmap

Removes an implicit map from the list of maps. Note that the unmap command does not remove explicit mappings (mappings created using the create_map command) between ports that might have been specified elsewhere in the Tcl setup.

Usage

unmap [-input | -output | -flop] ***spec_iof impl_iof***

Arguments

- -input | -output | -flop

  Restricts unmaps to inputs, outputs, or flop maps. Inputs include primary input ports and blackboxed output ports. Outputs include primary output ports and blackboxed input ports.

- ***spec_iof impl_iof***

  Specifies the names of specification and implementation input ports, output ports, or flops to be unmapped. The order is not important. Wildcard expressions can also be specified.

Description

By default, ports and internal flop outputs between designs whose names match are implicitly mapped. Implicit maps can be removed from the map list using the unmap command. For example, the following command removes the implicit map between ports spec.in47 and impl.in47.

```
unmap -input spec.in47 impl.in47
```

If using explicit maps, any implicit maps associated with their ports will be removed. For example, the following command will unmap all implicit maps (if they exist) between spec.in47-impl.in47 and spec.in48-impl.in48.

```
create_map -input spec.in47 impl.48
```

Wildcards can be used to specify names of flop maps, input ports and output ports. For example, the following command ensures that SLEC does not verify name-based output maps between the spec.dataValOut and impl.dataValOut ports. Note that name-based input maps between spec.dataValIn and impl.dataValIn are not be removed by this command.

```
unmap -output spec.dataVal* impl.dataVal*
```

unroll_loop_with_limit

Specifies the number of times a loop must be unrolled for a given filename:line number/label combination. Applies only to Verilog and SystemC design files read using the read_design or build_design commands.

Usage

unroll_loop_with_limit **-file** ***filename*** { **-line** ***line_number*** | **-label** ***label*** } **-limit** ***unroll_limit*** [-spec | -impl]

Arguments

- **-file** ***filename***

  Specifies the name of the file in which the loop exists.

- **-line** ***line_number*** | **-label** ***label***

  Specifies the line number or label associated with the loop.

  Note:
  The -label option can only be used with a SystemC design file.

- **-limit** ***unroll_limit***

  Specifies the number of times the specified loop must be unrolled.

- -spec | -impl

  Specifies the library containing the loop to unroll.

Description

The unroll_loop_with_limit command is used for loops whose termination condition cannot be determined at compile time. Such loops are referred to as data-dependent loops because their termination condition is dependent on data which will only be available at runtime.

In case the build_design command is unable to unroll a data dependent loop, compilation will fail. For such cases, call the unroll_loop_with_limit command before the build_design command to successfully compile the design.

For example, the following command unrolls the loop defined in line 23 of the file *fifo.cpp* 100 times.

```
unroll_loop_with_limit -file fifo.cpp -line 23 -limit 100 -spec
build_design -spec fifo.cpp
```

Likewise, if a label X_LOOP was being used to refer to the loop instead of the line number, the following commands would be used:

```
unroll_loop_with_limit -file fifo.cpp -label X_LOOP -limit 100 -spec
build_design -spec fifo.cpp
```

where, the file *fifo.cpp* contains a line with reference to the loop X_LOOP as follows:

```
X_LOOP: for (int x=0; x < port.read(); ++x)
```

verify

Verifies that the specification and implementation designs are equivalent or generates a counterexample.

Usage

verify [-mode *mode*] [-proof_depth *proof_depth*] [-stop_after *stop_after*] [-effort *effort*]

Arguments

- -mode *mode*

  Specifies the proof mode to use for verification. Legal values are bounded_proof, bug_hunt, and full_proof. Default is bounded_proof.

- -proof_depth *proof_depth*

  Specifies the maximum equivalent proof depth to verify if a full proof is not found before then.

- -stop_after *stop_after*

  Stops verification after completing the specified verification step. Legal values are setup, sim, and solve.

- -effort *effort*

  In the full-proof mode, this option specifies the effort to be applied in upfront simplification of the sequential problem, before attempting the output proofs. Legal values are low, medium, and high. Default is medium.

Description

Verifies that the specification and implementation designs are equivalent or generates a counterexample. Depending on the proof mode, the designs may be proven fully equivalent or proven equivalent only to the specified proof depth.

Stopping after setup or sim can be used to help debug problem setup before formal verification.

In the full-proof mode, SLEC attempts performing simplification of the designs being compared before attempting proofs on the outputs. In general, this upfront simplification aids the verification performance. However in some cases, it is possible that the simplification step could itself become a performance/memory bottleneck. In other cases, the verification performance could be improved by spending more effort on upfront simplification. If SLEC seems to be getting stuck/running out of memory in the full-proof mode, it is advisable to try running with the low and high effort levels.

Review the log files carefully for information about the results of verification.

view_waveform

Opens the specified waveform for viewing.

Usage

view_waveform [-spec | -impl] [-slec | -tb] [*trace_dir*]

Arguments

- -spec | -impl

  Specifies whether to open the specification or implementation waveform. If you do not specify this argument, SLEC opens both waveforms.

- -slec | -tb

  Specifies which waveform to open. Use -slec to open the SLEC-generated waveform. Use -tb to open the testbench-generated waveform. If you do not specify this argument, the SLEC-generated waveform is opened by default.

  Note:
  SLEC generates testbenches only when the following conditions are true: a falsification is found, and the global variable generate_testbenches is enabled.

- *trace_dir*

  Specifies the counterexample trace directory. SLEC resolves the trace directory as follows:

  - If *trace_dir* is not specified, then:
    - If the current directory is a trace directory, select the current directory.
    - If the current directory is not a trace directory, then:
      - If config_trace_files was executed with the -directory *dirname* argument, select the latest trace generated in *dirname*.
      - Otherwise, select the latest trace generated in *workdir*.
  - If *trace_dir* is a relative path, select *./<trace_dir>*.
  - If *trace_dir* does not begin with "trace_" prefix, select *./trace_<trace_dir>*.
  - If *trace_dir* is an absolute path, select *<trace_dir>*.

  See Viewing Waveforms in the *SLEC User's Manual* for a full explanation of how SLEC resolves the trace directory.

Description

The view_waveform command invoke the appropriate waveform viewer based on the waveform format: Novas Verdi for VCD or FSDB formats, and Visualizer for the QWAVE format. The SLEC engines, such as simulation-based validation, counterexample replay, concolic engine, and formal engines, can generate multiple counterexamples on a given run. SLEC may also generate multiple counterexamples when the global variable stop_at_first_falsification is 0. Check the contents of *slec.log* to determine trace directories for counterexamples. You can specify the counterexample directory in the *trace_dir* field above.

write_db

Writes the database to a file.

Usage

write_db ***file***

Arguments

- ***file***

  Specifies the name of the database file to create.

Description

The write_db command is used to write a design database file. The database should be written after both designs have been built and before any clock, constraint, map, or verify command is applied.

Consider the following example. This example writes the design database file to the *file.db* file.

```
write_db file.db
```

Database files can be written using relative or absolute pathnames. For example,

```
write_db ./sub-dir/file.db
```

Using the write_db command to save a database can speed up run time. In this example, the design source files are read if the DB file *design.db* does NOT exist. After the design files are read in, the design is written to the DB file, any subsequent runs will read the DB file instead of the design source files. Reading a DB file is typically faster than reading design source files.

If using this example, remember to delete the DB file if the source code is modified; otherwise the old DB file will be read instead of the new code.

```
set db_name design.db

if {[file exists $db_name]} {

    # Read the DB file
    read_db $db_name

    }  else {

         # Else build design and write a new DB file
```

```
        build_design -spec -f spec_files
        build_design -impl -f impl_files
        write_db $db_name
    }
```

Chapter 3
Global Variable Reference

This chapter describes the global variables used by SLEC. Use the set_global command to set values to global variables.

alert_color

Text color for alert messages such as warnings and errors on color-capable terminals. Legal settings are black, blue, green, cyan, red, purple, brown, gray, dark_gray, light_blue, light_green, light_cyan, light_red, light_purple, yellow, white, and off (to disable the feature). Default setting is light_red. Colorization is automatically disabled on non color capable interfaces such as output redirection and output piping.

Description

Alert messages such as warnings and errors may be displayed in a unique color to highlight their presence. This variable can be set to customize the desired color, depending on the terminal environment.

allow_int_to_enum_conversion

Allows conversion of int to enum types. Conversion of int to enum is disallowed by default. Legal settings are 0 or 1. Default is 0.

Description

Conversion of int to enum is disallowed as per ANSI C standard. However, for backward compatibility with Kernighan and Ritchie C, this global can be turned on by setting it to 1.

Note:
This global works only with SystemC designs.

assert_all_sva_assumes

Directs SLEC to treat SVA assumptions as asserts.

Description

This affects the behavior of the check_properties and assume_properties commands.

assume_all_sva_asserts

Directs SLEC to treat SVA asserts as assumptions.

Description

This affects the behavior of the check_properties and assume_properties commands.

assume_array_indices_in_range

Enables optimization for array indexing. Legal settings are 0 or 1. Default setting is 0.

Description

By default SLEC checks to see if array indices are out of bounds. If they are out of bounds, then the result is an unknown value (X). Setting the global assume_array_indices_in_range disables array bound checking and tells SLEC to synthesize the design assuming that the index can address only the size of the array. Note that this global should be set only if the user can guarantee that all array indices will be in range.

async_site_setup

Specifies the Tcl script that defines how to submit a job.

Description

This variable contains the filename for a Tcl script that defines how to submit asynchronous jobs.

async_subsession_callback_period

Specifies the time between asynchronous job execution.

Description

When launching SLEC subsessions with run_subsession, you can specify the -async argument to place the job in the queue for asynchronous execution. The join_async_subsession command launches the queued jobs according to the distributed processing configuration. The global async_subsession_callback_period determines the time interval (in seconds) between executions of the periodic callback provided to the join_async_subsessions command. See Subsession-level Distributed Processing in the *SLEC User's Manual*.

attempt_clock_map_verification

Attaches a transactor with a posedge flop and a negedge flop to the specification and implementation ports. Legal values are 0 and 1. Default: 0.

Description

By default, SLEC ignores any mapping attempts on clock output ports. When a clock root fans out to a primary output port or a blackbox input port that does not stop at a flop or a latch, it is referred to as a clock output port.

Setting this global to 1 will instruct SLEC to attach a transactor with a posedge and a negedge flop to the specification and implementation ports, allowing the user to observe the clock consequence indirectly.

binary_sim_mode

Enables binary simulation mode which does not perform conservative X value simulation. Legal settings are 0 or 1. Default setting is 0.

Description

Default 3 valued simulation can sometimes give false negatives. An "X" value may appear on an output and cause a falsification to occur when the "X" is really a "don't care" value. Another reason for a false X value to appear on an output being compared, is that SLEC's internal simulation engine cannot handle bit wise X values i.e. a 4 bit value of "101X" is treated to be "XXXX". In order to avoid such false negatives from SLEC, the global binary_sim_mode can be turned on. This ensures that during the simulation of the specification and implementation designs during normal operation, no Xs appear at any signals.

The various sources of Xs (or Zs) in simulation are as follows:

- Xs/Zs in the design: Enabling binary_sim_mode makes simulation assign random values to such Xs/Zs during both reset simulation and simulation of normal operation.
- Xs in the reset state: Enabling binary_sim_mode makes simulation of normal operation assign random values to such Xs. Note that this does not affect the reset state for the solver.
- Xs on unconstrained inputs and black box outputs during reset simulation: If the global value_on_input_pins_during_reset is set to "X" (which is its default value), then during reset simulation SLEC drives Xs on inputs and black box outputs. This behavior is not affected by enabling binary_sim_mode

bit_level_solver_only

Specifies that the formal engines (in verify) should only use the bit-level solvers for the current verification run. For details, please refer to the capacity Application Note (AN005). Legal settings are 0 or 1. Default setting is 0.

Description

SLEC uses a combination of techniques to formally verify whether the specified pair of designs are equivalent. These include word level solvers and bit level solvers. In case one or both of the designs is

bit level heavy, then using the global bit_level_solver_only can improve the SLEC runtime by avoiding the use of the word level solvers. The ratio of the gates to nodes, as reported in characteristics.log is a good measure of whether a design is bit level heavy or not; with a low ratio indicating that the design is bit level heavy.

Exercise caution when enabling this global; it is possible that a few intermediate problems which are key for the verification can be solved only by word level solvers.

cex_read_only_store_location

Defines the location for retrieving counterexample files from the previous run. This path can be an absolute path or a relative path from the current working directory. Other legal settings include homedir and none. Default: none.

Description

Before running simulation-based validation, SLEC loads counterexamples from the previous run. If it discovers any falsifications, it handles them just as any other falsification. This feature enables quick debugging of counterexamples. No new counterexamples are stored in cex_read_only_store_location.

If both cex_store_location and cex_read_only_store_location are set to the same location, SLEC issues a warning and treats cex_store_location as none. In this case, SLEC reads counterexamples but does not store new ones.

cex_store_disk_usage

Determines the maximum size (in MB) of the counterexample storage directory.

Description

If SLEC is configured to save counterexample files, cex_store_disk_usage determines the maximum size of the counterexample storage directory. If it exceeds this size while SLEC is writing a new counterexample file, SLEC removes older counterexamples from the directory. To determine which to remove, it applies a Least Recently Used (LRU) policy. The cex_store_eviction_rate global determines the percentage of counterexamples SLEC will remove.

cex_store_eviction_rate

Defines the percentage of counterexamples to remove when the counterexample storage directory exceeds its maximum allowable size. Legal range of values is 0 to100. Default is 30.

Description

If configured to save counterexample files, SLEC stores them in the cex_store_location directory. When this directory reaches its maximum allowable size, as specified by cex_store_disk_usage, it removes counterexamples from the directory following a Least Recently Used (LRU) policy. The cex_store_eviction_rate defines the percentage of counterexamples to remove each time this condition occurs.

cex_store_location

Defines the location for storing and retrieving counterexample files from the previous run. This path can be an absolute path or a relative path from the current working directory. Other legal settings include homedir, workdir, and none. Default: workdir.

Description

When SLEC finds a design difference, it generates a counterexample which it replays on subsequent runs. The cex_store_location determines the location where SLEC saves the counterexample files. Other acceptable values include:

- homedir : points to the *~/.slec/cex_store/* directory.
- workdir : points to the *<current working directory>/cex_store/* directory. This is the default value.
- none : does not store counterexample files.

cex_store_replay_count

Specifies the maximum number of counterexamples to replay from the counterexample storage directory. Legal setting is any positive integer. Default is INT_MAX.

Description

Before running simulation-based validation, SLEC loads counterexamples from the previous run. If it discovers any falsifications, it handles them just as any other falsification. This feature enables quick debugging of counterexamples.

The cex_store_replay_count value global defines the maximum number of counterexamples to replay. The default value is INT_MAX, resulting in practically unlimited replays. If cex_store_replay_count is set to 0, then SLEC does not replay counterexamples from the counterexample storage directory.

check_vacuous_proofs

Check formal proofs for vacuity.

Description

Checks formal proofs for vacuity of all maps containing a valid condition. If the setup contains global assumptions, SLEC checks all maps. It sets up a formal coverage objective to verify that the assumptions are not all false forever. Vacuity checking can incur additional runtime penalty. Upon completion, SLEC reports a warning with the number of proofs that were vacuous and generates a detailed report in *results.log* in the *work* directory.

configure_ac_probe_include_line_number

Appends the line number of the ac::probe_map() function call in the name of the signal created as a result of the call. Possible values are 0 and 1. Default is 0.

Description

The configure_ac_probe_include_line_number global appends the line number of the ac::probe_map() function call in the name of the signal created as a result of the call. By default, the line number is not included in the name because if any changes are made to the source code, especially until the ac::probe_map() function is run, it will change the name of the hook.

Consider the following example:

```
template <typename T>
void test( temp <T> t1 )
{
    int b = t1.a;
    ac::probe_map( "int_probe_map", b ); // line 5
}
```

By default, the name of the signal, created as a result of the ac::probe_map() function call, is test()#0@ac_probe_int_probe_map. If the global is set to 1, name of the signal will be test()#0@ac_probe_int_probe_map_ln_5.

configure_ac_probe_long_function_names

Includes the template parameters and the argument types of a function in the name of the signal created as a result of the ac::probe_map() function call. Possible values are 0 and 1. Default is 0.

Description

The configure_ac_probe_long_function_names global includes the template parameters and argument types of a function in the name of the signal created as a result of the ac::probe_map() function call. By default, the name of the signal does not include the template parameters and argument types.

Consider the following example:

```
template <typename T>
void test( temp <T> t1 )
{
    int b = t1.a;
    ac::probe_map( "int_probe_map", b );
}
```

By default, the name of the signal, created as a result of the ac::probe_map() function call, is test()#0@ac_probe_int_probe_map. If the global is set to 1, name of the signal will be test<int>(temp<int>)#0@ac_probe_int_probe_map.

convert_flopmaps_to_intermediate_maps

When set to 1, SLEC automatically converts all flop maps to intermediate maps. To disable automatic conversion, set this global to 0. Legal settings are 0 and 1. Default setting is 1 for supported verification modes. For a list of supported verification modes, refer to the set_verification_mode command.

Description

Note:
Starting with SLEC 6.1, support for flop maps is gradually being phased out. Calypto recommends using intermediate maps instead of flop maps.

For supported verification modes, SLEC automatically converts all user and HLS specified flop maps to appropriate intermediate maps. In situations where conversion is not desired, set the convert_flopmaps_to_intermediate_maps global to 0. Note that this global must be specified after the set_verification_mode command has been executed.

For more information on intermediate maps, refer to Application Note #13: Using Intermediate Maps in SLEC.

cpt_error_trace_limit

Specifies the trace limit for error messages issued by the C++ elaboration engine.

Description

By default, the trace limit for warning messages is 5, that is, it backtracks the origin of the message up to five function calls.

cpt_warn_trace_limit

Specifies the trace limit for the warning messages issued by the C++ elaboration engine.

Description

By default, the trace limit for warning messages is 0, that is, no trace is reported for warning messages.

cut_at_mapped_flops

Enables cutting of all flops mapped between designs. Legal settings are 0 or 1. Default setting is 0.

Description

By default flops are not cut. Flop inputs are latched on appropriate clock edges, and these values drive the flop outputs. Conversely, when flops are cut, matching symbolic values are used to drive the flop outputs instead of the latched input values. This relaxation removes the mapped state space from both designs and improves the ability to find full proofs and deeper bounded proofs. However, because flop outputs are unconstrained, a counterexample may be produced which depends on unreachable state space.

In order to specify that SLEC cut at a subset of flops involved in certain flop-maps, use the -cut option in "create_map -flop".

designware_library_path

Specifies the path to the Designware library installation.

Description

This global is required for SLEC-generated testbenches to work seamlessly on designs that use Designware components. This path is used in the makefile to pick up the Designware library elements automatically. Here is an example:

```
set_global designware_library_path  /user/lib/dware
```

dynamic_alloc_chunk_size

Specifies the maximum size of memory which will be allocated for a malloc/new call. Legal setting is any positive integer. Default setting is 1024.

Description

The SLEC C++ frontend supports dynamic allocation of memory using new or malloc calls. The maximum size of memory units allocated for a single new or malloc call is restricted to 1024 units. The tool will error out if an allocation larger than 1024 is requested.

The maximum size of memory that can be allocated for single dynamic memory allocation call can be controlled by setting this global to new value.

enable_concolic_engine

Boolean variable that specifies whether to use the concolic engine in the verification flow. Default: false.

enable_floating_point_support

Boolean variable that specifies whether to enable floating point support. Default: true.

enable_hierarchy_synthesis

Instructs the build_design command to synthesize hierarchies for all SystemC/C++ functions that have the appropriate pragma as follows: #pragma map_to_operator [CCORE]. This also enables the hierarchical-verification flow for SystemC/C++ designs. The legal settings are 0 or 1. Default setting is 0.

Description

To improve SLEC's performance for SystemC/C++ designs, a hierarchical verification strategy can be used if the following conditions are met:

- The specification and implementation designs have a throughput of 1.
- The functions to be marked for hierarchical synthesis have identical interfaces.
- The function argument is not an array of pointers, a union or a struct with union data-members.
- A pointer to an object is not passed twice as an argument to the hierarchy function.
- The function does not access a static or a global variable.
- The function does not have a SystemC wait() statement in it.
- The function is not recursive.
- The function returns a void or basic datatype. For example, int or sc_int.

Consider the following top-level function (corresponding to SC_METHOD).

```
void do_processing(...)
{
   ...
   int rval = do_complex_processing(arg1, arg2);
   // Use rval downstream
   ...
}
```

Assuming the above mentioned conditions have been met, a hierarchical verification strategy can be deployed as follows:

1. Insert the following pragma for the function do_complex_processing in the C++ source code.

   ```
   #pragma map_to_operator [CCORE]
    int do_complex_processing(int formal1, char formal2)
   {
    ...
   }
   ```

2. Before using the build_design command, use the global enable_hierarchy_synthesis to instruct SLEC to synthesize the functions that have the pragma design as hierarchies.

3. The design database created by the build_design command will have a hierarchy corresponding to the do_complex_processing function.

   To verify, use the following find commands to look for the corresponding hierarchy and hierarchy-instances respectively.

```
slec> find -spec -hier -module
spec_wrapper do_complex_processing

slec> find -spec -hier -inst -of_cell_type do_complex_processing
spec.do_complex_processing_0_inst_1
```

4. Perform a SLEC run to verify the function do_complex_processing in the specification and implementation designs.

```
set_global enable_hierarchy_synthesis 1
build_design -spec ...
build_design -impl ...

# Setup for the verification of the function-pair
...

verify
```

5. Perform another SLEC run to verify that the functionality of the do_processing function in the specification and implementation designs are equivalent.

```
set_global enable_hierarchy_synthesis 1
build_design -spec ...
build_design -impl ...

# Setup for top-level verification with the function-hierarchies
# blackboxed
#
create_black_box -inst spec.do_complex_processing_0_inst_1
 create_black_box -inst impl.do_complex_processing_0_inst_1
...
verify
```

enable_ll1_engine

Boolean variable that specifies whether to use the low-effort level-1 engine in the verification flow.

enable_ll3_engine

Boolean variable that specifies whether to use the low-effort level-3 engine in the verification flow.

enable_low_effort_induct_init_check

Boolean variable that specifies whether to use low-effort induct init checks (induction base case) in the verification flow.

enable_modular_interfaces

Enables the compilation of SystemC designs which contain modular interfaces. Legal settings are 0 or 1. Default setting is 0.

Description

By default, the tool does not compile SystemC designs which contain modular interfaces. To enable the compilation of such designs the global enable_modular_interfaces should be set to 1.

This global should be set before the build_design command is issued. Valid values of this global are 1 and 0 and it has a default value of 0.

enable_parallel_solverloop

This global will turn ON parallelization for solvers. Legal settings are 0 or 1. Default setting is 0.

enable_pens

Enables checking for potentially equivalent nodes (PENs) internal to the design. Legal settings are 0 or 1. Default setting is 1.

Description

The formal solvers divide and conquer the verification by finding equivalent internal signals and proving them. This often simplifies the verification problem and reduces runtime. However, in some cases, proving the intermediate equivalence can have the opposite effect, leading to longer runtimes. If you observe slow runtimes, set this global to 0 and re-run verification. The default value is 1 so intermediate PEN checking is enabled.

enable_seqsat_async_subsessions

Enables distributed processing on sequential engines LL3 and HL3. Default: 0.

enable_solver_async_subsessions

Enables distributed processing on combinational engines LII, HII, LL1, HL1, HL2, and BEC. Default: 0.

exit_on_error

Forces SLEC to exit a Tcl script rather than continue debugging in case of premature termination by a script error. Legal settings are 0 or 1. Default setting is 0.

Description

When this global is set to 0, SLEC traps any script errors and presents the user with an interactive shell prompt for debugging. When this global is set to 1, if a scripting error is encountered, SLEC exits without prompting the user.

expert_system_processes

Specifies the number of OS processes to apply for proof strategy exploration in LL3 bit-level solvers. The expert system uses parallel processes to explore different proof strategies, and terminates when any one of the proof strategies is successful in solving the problem. Legal values: 1 or greater. Default: 5.

Description

The expert_system_processes global specifies the number of OS processes to apply for proof strategy exploration in LL3 bit-level solvers. In most cases, applying the expert system can significantly reduce the time taken to solve a problem. However, as parallel processing leads to increased computational resource utilization, it is possible that some machines might slow down, leading to increased run time. To achieve optimal run time savings, set to the number of CPU cores available.

external_case_split_async_subsessions

Boolean variable that indicates whether or not to spawn asynchronous subsessions for external case splits. Default: false.

find_invariants

Specifies that SLEC attempt to find extra invariants during upfront analysis in the bounded_proof mode, to help improve SLEC's performance. Legal settings are 0 or 1. Setting this global to 1 effectively enables the Level-1 proof engine in SLEC. Default setting is 0.

Description

In the full_proof mode, SLEC attempts finding and utilizing invariants by default. However finding such invariants can be compute intensive. Therefore SLEC does not attempt finding them by default in the bounded_proof mode. It is recommended that this global should be turned on when a large proportion (for example, more than 75%) of the flops in the specification and implementation designs are mapped. Under such circumstances, it is expected that turning on the global will improve SLEC's performance.

flop_checking_at_reset

Switches between three different modes for comparing mapped flop states after reset. Legal settings are concrete, symbolic, or relaxed. Default setting is symbolic.

Description

The flop_checking_at_reset global controls how mapped flops are compared after reset. The three modes are concrete, symbolic (the default), and relaxed. In the concrete mode, the spec and impl flop values must either be 1-1 or 0-0 to avoid falsification. In the symbolic mode, they must be 1-1, 0-0, or X-X. In the relaxed mode, the flop values may be any of 0-0, 1-1, X-X, 0-X or 1-X. In the last two cases the X value is

replaced by the corresponding concrete value (0/1). Note that these checks only apply for flop maps with zero latency for both spec and impl (including by-name pairs).

generate_testbenches

Enables the creation of testbenches for any counter-examples that were found. Legal settings are 0 or 1. Default setting is 1.

Description

Turn off this global if you are not interested in generating testbenches for the counterexamples that SLEC finds.

host_setup_configuration

Specifies the command for launching remote jobs. Set this global according to the grid system in use.

Description

The host_setup_configuration global specifies a job submission command for launching remote jobs. Consult with the documentation for your grid system to determine the specific syntax for job initiation.

The job submission command must submit the job for execution and return without waiting for the job to start or execute. For most systems, this will take a few seconds, at most.

Do not use any interactive options (you watch the command run in your terminal) or waiting options (the submission command does not return until the submitted job completes).

You can access the remote job counter in your command with the syntax: "%RJC". (For example, if you wanted to give each job a unique job name or description or have any other reason.)

Each job submitted with your job submission command will submit a job which consists of a *workdir* /CreateRemoteAsyncWorker/run%RJC.sh script, appended as the last argument to your specified command. The output of the job submission command (stdout and stderr) is saved to *workdir*/CreateRemoteAsyncWorker/run%RJC.sh-submit.log. For more information on the Working Directory, refer to SLEC Work Directory.

For example, the following command prepares SLEC for running jobs with Univa® Grid Engine® and creates a separate log for each worker:

```
set_global host_setup_configuration "qsub -l h_rt=infinity -V
  > qsub_out_%RJC.log"
```

ignore_intermediate_points

Skips the proof of intermediate signals. Legal settings are 0 or 1. Default setting is 0.

Note:
This command is now deprecated. See enable_pens.

Description

The formal solvers divide and conquer the verification by finding equivalent internal signals and proving them to make the final output proofs easier. However, the proof of internal signals can sometimes consume more resources than just proving output cones only. If the equivalence takes a very long time, it is recommended that this global be set to 1. The default value is 0 so intermediate points are proven.

impl_output_latency

Specifies the latency of the implementation design for name-based output maps. Legal setting is any non-negative integer. Default setting is 0.

Description

In the context of sequential equivalence checking, the latency of an output port or a black box input port indicates the cycle in a transaction, at which it needs to be checked for equivalence with the corresponding output.

For name-based output maps, the globals spec_output_latency and impl_output_latency are used to specify the latencies of the specification design outputs and the implementation design outputs. By default, it is assumed that a design produces mapped outputs starting with the first cycle so the default values of these globals is 0.

Note:
These globals only specify the latencies for output ports or blackboxed input ports that are implicitly mapped by name.

For example, if both the specification and implementation designs have the output ports outA and outB, then SLEC will verify these output-maps by default, unless an explicit unmap command has been specified for any of these maps.

```
create_map -output spec.outA impl.outA
create_map -output spec.outB impl.outB
```

In the above maps, the latencies of both the specification and implementation outputs are zero. If the comparison needs to be done with a latency adjustment, then the globals spec_output_latency and impl_output_latency can be used. For example, if the specification design outputs in cycle 1 of every transaction needs to be verified against the implementation design outputs in cycle 2 of every transaction, then the global settings should be:

```
set_global spec_output_latency 1
set_global impl_output_latency 2
```

Note that the above settings are equivalent to explicitly specifying the following create_map commands:

```
create_map -output [create_waveform -sample_start 1 spec.outA]
                   [create_waveform -sample_start 2 impl.outA]
create_map -output [create_waveform -sample_start 1 spec.outB]
                   [create_waveform -sample_start 2 impl.outB]
```

For cycle-accurate designs it is neither necessary nor desirable to specify the real output latency. If the specification and implementation designs can be compared every cycle, then this global should be

allowed to default to 0. For optimum design capacity, the smallest possible value for output latency should be used.

Note:
This global does not affect flop maps or explicit output maps.

impl_throughput

Specifies the throughput of the implementation design, in cycles. Design throughput of <*n*> cycles indicates that the design consumes inputs and produces meaningful outputs every <*n*> cycles. The default throughput is 1. Legal setting is any positive integer. To specify the throughput of the specification design, use the global spec_throughput.

Description

The globals spec_throughput and impl_throughput specify the number of cycles required by the specification and implementation designs to transition to a state of equivalence where they:

- Consume mapped inputs (provided by SLEC)
- Produce mapped flops or outputs that need to be verified by SLEC

Consider a typical scenario where an untimed C++ model is compared with its RTL implementation. Assuming the C++ model consumes inputs every cycle and produces an output immediately. The global spec_throughput would be set as follows:

```
set_global spec_throughput 1
```

Now, assuming the corresponding RTL takes 200 cycles to produce equivalent output and is ready to accept inputs corresponding to the second cycle of the C++ model in the 201st cycle, the global impl_throughput would be set as follows:

```
set_global impl_throughput 200
```

For cycle-accurate designs it is neither necessary nor desirable to specify the real throughput. If the specification and implementation designs can be compared every cycle, then this global should be allowed to default to 1. For optimum design capacity, the smallest possible value for throughput should be used.

For example, the designs being compared could be a RTL block which implements an image-processing transform and another cycle-accurate version of it. From the designer's perspective, the throughput could be several thousands of cycles that the block takes to complete the transform. However, given that the designs are cycle-accurate, they can be compared with both the throughputs set to 1.

infer_memory_map_latency

Instructs the verify command to detect the latency in mapping between the memories. Legal settings are 0 or 1. Default setting is 0.

Description

Once this global is set, verify will attempt to find the latency interval with which the memories specified in the create_memory_map command map to each other. These latency numbers need to be fed back to the create_memory_map command to run verify.

large_throughput_optimization

Enables SLEC to perform optimizations specific for large throughputs. Legal settings are 0 or 1. Default setting is 0.

Description

If either of spec_throughput or impl_throughput is large, then the verification complexity increases commensurately. Setting this global enables SLEC to perform optimizations for large throughputs. If the product of the gate count and throughput is greater than one million for the specification or implementation design, this optimization might prove more efficient.

Note:
Optimizations might reduce both combinational and sequential complexity for verification.

ll1_problem_time_limit

Integer variable that specifies the per-problem time limit for the low-effort level-1 verification engine.

The ll1_problem_time_limit variable takes a value in seconds to determine the per-problem time limit. A value of 0 indicates no limit. The default value is 20.

ll3_problem_time_limit

Integer variable that specifies the per-problem time limit for the low-effort level-3 verification engine.

The ll3_problem_time_limit variable takes a value in seconds to determine the per-problem time limit. A value of 0 indicates no limit. The default value is 30.

map_x_as_symbolic

Matches X values from structurally similar regions of the specification and implementation designs, and replaces them with mapped symbolic inputs to avoid consistency-check warnings or falsifications caused by uncorrelated X or Z comparisons. Legal values are 0 and 1. Set this global to 1 to enable it and 0 to disable it. By default, this global is set to 0.

Description

SLEC's formal solvers only reason the binary values 0 and 1. It does no formal reasoning for X values. If X values are explicitly specified in a design, the default behavior causes them to be replaced by unconstrained symbolic values. This default behavior can propagate uncorrelated values to outputs and result in false counterexamples. However, if the specification and implementation designs have

structurally similar origins of the X values, enabling this global can help correlate them and avoid a false counterexample. Consider the following example:

```
// Specification Design
case (sel)
    2'b00:   mux_out <= ...
    2'b01:   mux_out <= ...
    default: mux_out <= 2'bx;

// Implementation Design
case (new_sel)
    3'b000:  mux_out <= ...
    3'b101:  mux_out <= ...
    default: mux_out <= 2'bx;
```

In this example, both the specification and implementation design conditionally assign a 2-bit X value to a signal mutually called mux_out. When this global is enabled, SLEC automatically recognizes the matching hierarchical names of the affected signals and the matching bitwidth, causing it to map both X sources to a symbolic input. The resulting auto-injected blackbox is reported in the *mapping.log* file. For example:

```
### List of input-maps
    create_map -input spec.explicit_X_0_1_0_mux_out_bbox.bbox_out
                     impl.explicit_X_0_1_0_mux_out_bbox.bbox_out
```

To avoid unintended over-constraining of dissimilar X values, this matching heuristic is conservative. As a result, not all values will necessarily get replaced with mapped symbolic inputs.

map_z_as_symbolic

Matches Z values from structurally similar regions of the specification and implementation designs, and replaces them with mapped symbolic inputs to avoid consistency-check warnings or falsifications caused by uncorrelated X or Z comparisons. Legal values are 0 and 1. Set this global to 1 to enable it and 0 to disable it. By default, this global is set to 0.

Description

SLEC's formal solvers only reason the binary values 0 and 1. It does no formal reasoning for Z values. If Z values are explicitly specified in a design, the default behavior causes them to be replaced by unconstrained symbolic values. This default behavior can propagate uncorrelated values to the outputs and result in false counterexamples. However, if the specification and implementation designs have structurally similar origins of the Z values, enabling this global can help correlate them and avoid a false counterexample. Consider the following example:

```
// spec design
   mymod  inst1  (.in1(signal1), .in2(/*unconnected*/), .out(signal5))

// impl design
   mymod  inst1  (.in2(/*unconnected*/), .in3(signal3), in4(signal4),
                  .out5(signal5), .out6(signal6));
```

In this example, both the specification and implementation designs have an instance called inst1 with a port in2 that is unconnected (undriven). With this global enabled, SLEC will inject a mapped symbolic input on the unconnected ports because:

- their hierarchical port names match
- their bitwidths are the same
- they are both undriven

The resulting auto-injected blackbox is reported in mapping.log. For example:

```
### List of input-maps
 create_map -input spec.unconnected_Z_0_1_0_in2_bbox.bbox_out
       impl.unconnected_Z_0_1_0_in2_bbox.bbox_out
```

To avoid unintended over-constraining of dissimilar Z values, this matching heuristic is conservative. As a result, not all values will necessarily get replaced with mapped symbolic inputs.

maximum_iter_count

Specifies the maximum number of iterations for unrolling immediate loops in a design description. Legal setting is any positive integer. Default setting is 1024.

Description

An immediate loop is a loop which contains no synchronizing constructs. It executes within one time step. Such loops are unrolled by the SLEC compiler. By default, there is no limit on the number of iterations for a loop. Setting this global to an integer value establishes a limit on the number of iterations. If a loop exceeds the defined limit, then an error message (CPT-MICR) will be generated. If you is aware of an actual upper limit, use the unroll_loop_with_limit" command to unroll the loop.

maximum_recurse_count

Specifies the maximum depth of recursion for function calls in a design description. Legal setting is any positive integer. The default setting is 100.

Description

Recursive function calls are unfolded and inlined by the SLEC compiler. Setting this variable to an integer value establishes a limit on the depth of recursion. If the limit is exceeded, then an error message is generated.

max_local_async_workers

Specifies the maximum number of worker processes to run on the local (master) machine. A setting of 0 indicates that the master will not perform verification tasks; it only collates results from worker processes. Default is 1.

Note:
Before increasing this value, ensure you have sufficient CPU resources and licenses available.

max_remote_async_workers

Specifies the maximum number of worker processes to run on remote machines. By definition, this value also specifies the maximum number of worker processes with which the master process can be associated. Default is 0.

Note:
For distributed computing environments, increase this value based on your specific system configuration and number of licenses available.

message_wrap_at

Specifies the number of characters per line at which the generated messages should be wrapped. Legal settings are any positive integer. Default setting is 79.

Description

Use this global to make SLEC-issued messages be formatted based on the size of the window.

signal_files_path

Specifies the path to the directory where SLEC saves Novas Verdi and Visualizer signal files.

Description

You can specify this global as a relative path from the current working directory or as an absolute path. If the specified directory does not exist, an error is reported. If you do not set this global, SLEC save the signal files in the respective *trace_** directory.

For example, the following command configures SLEC to save the signal files in the current directory.

```
set_global signal_files_path .
```

novas_tool

Determines the name of the executable invoked for Novas view_waveform targets. Default: Verdi.

novas_tool_switches

Determines the arguments passed to the executable invoked for Novas view_waveform targets. Default: -ssv -ssy -ssz.

optimize_for_highly_constrained_inputs

Enables SLEC to perform a low cost BDD sweep of the PEN problem netlist to find additional equalities. Legal settings are 0 or 1. Default setting is 0.

Description

This global is useful when verifying design inputs that are highly constrained. Constraints on design inputs are typically propagated to internal nodes using constant propagation. However, sometimes when the resource limit is exceeded (depending upon the hardness and width of the arithmetic operators), the constraints may not be propagated deep enough into the design. As a result, many internal equivalences might not be discovered. Since PEN problems are usually significantly smaller in node/gate count compared to the node/gate count of the overall design, and often the design input constraints do propagate to PEN inputs, a low cost BDD sweep of the PEN problem netlist can find additional equalities that might not have been exposed on the overall netlist.

```
set_global optimize_for_highly_constrained_inputs 1
```

osci_compliant_initial_value

Enables SystemC datatypes to initialize to their OSCI-defined initial values rather than X values. Legal settings are 0 or 1. Default setting is 1.

Description

Design registers and local variables normally initialize to X values. However, for 2-valued SystemC datatypes, SystemC simulation semantics specify that these datatypes are initialized to 0 in their constructors. SystemC datatypes sc_int<N>, sc_uint<N>, and sc_bv<N> all initialize to 0, while C++ built-in types, such as bool and int initialize to unspecified values.

If osci_compliant_initial_value is ON, SLEC follows the SystemC simulation semantics and initializes all SystemC datatypes to their OSCI-compliant values. As a consequence, registers are initialized to their OSCI-compliant values as their power-on values. Local and non-register values are also initialized to OSCI-compliant values. Otherwise, registers and variables are initialized to X values. The default is ON. This global must be set before the build_design command is issued.

output_problem_time_limit

Limits the amount of time spent by a solver on each output problem, allowing one to bypass hard solver problems. Time limit is specified in seconds. Default setting is 0 which means no limits are set on time taken by a solver to solve an output problem.

Description

Multiple solvers may be invoked on an output problem at various stages in SLEC, such as level 1 proof engine, each transaction of BEC (level 2 proof engine), and level 3 proof engine. The time limit, in seconds, applies to an individual solver call, and hence the real time taken per problem can be a multiple of the specified value.

ovl_library_path

Specifies the path to the OVL library installation.

Description

To generate testbenches for OVL instances, set the ovl_library_path global to point to the directory containing the simulation models for the OVL assertions. This path is used to automatically fetch the simulation models for the OVL assertions.

prune_unmapped_logic

Specifies whether SLEC should perform an upfront removal of dead logic so logic is not fanning out to any mapped outputs. Legal settings are 0 and 1. Default setting is 1.

Description

When this global is set to 1, SLEC performs an upfront removal of signals which are not fanning to any mapped outputs. Set this global to 0 if signals are missing in the waveform.

An early removal of dead logic implies that consistency checks will not be performed on this part of the circuit. In addition, the waveforms generated by SLEC, for both simulation-based validation and counterexamples, will not include signals in dead logic.

replace_xz_with_constant

Replaces X and Z values in the design netlist with a constant value. Legal settings are -1, 0 or 1. Default setting is -1.

Description

SLEC's formal solvers can reason only in the binary domain: 0 and 1 values. It does no formal reasoning for X and Z values. By default, if these values are explicitly specified in a design, they are replaced by symbolic values and treated similar to unconstrained primary inputs by default. Sometimes these uncorrelated values propagate to outputs and result in false counterexamples.

Replacing all X and Z values with constant 0 and 1 values minimizes uncorrelated propagation, but this may have a side effect of potentially missing a genuine difference between the two designs. The default value is -1, which means do not replace netlist values.

Note that this global does not affect the reset state of flops. Any flop not set during reset begins in an X state which may propagate to primary outputs. Reset state may be explicitly set using the set_reset_value command. This global must be set before the build_design command is issued.

respect_async_reset

SLEC accurately models the asynchronous-reset behavior of flops when this global is set to 1. By default, SLEC models asynchronous-reset as synchronous-reset. Legal settings are 0 or 1. Default setting is 0.

Description

By default, SLEC models asynchronous-reset behavior as synchronous-reset while performing verification. If however, the asynchronous-reset modeling is desired, this global can be set to 1. Note that with this global set to 1, SLEC will perform Sync-Async modeling of flops i.e. the flop fed by the asynchronous-reset will continue to be modeled as if the reset is synchronous though all of the logic fed by the flop will be modeled as if the flop has an asynchronous reset. As a result, while the overall design will now behave as if the flop had an asynchronous-reset, the two specific areas where the synchronous-reset modeling will continue to happen are:

- The waveform of the signal directly fed by the flop.
- Any flop-maps or output-maps directly involving the flop.

show_wildcard_expansion

Show wildcard expansions and substitutions in commands. Legal settings are 0 or 1. Default setting is 0.

Description

When set to 1, wildcards (*) are automatically expanded and substitutions are individually executed. The resulting series of expanded commands are logged in the slec.cmd file, slec.log file and displayed on screen. By default, the resulting expanded commands are only logged in the *slec.cmd* file.

select_address_slice_number

Controls the bits from the address bus of the original memory that are used to address the abstract memory. Legal settings are 0 or higher. Default setting is 0.

Description

The abstract memory is smaller than the original memory and hence requires less bits for addressing. Selection of these bits can be directed by this global. Multiple runs of verify can be done with different values of the global to ensure each address bit of the original memory is used in some invocation of verify. Setting the value of the global to 0 selects the bits adjacent to the fixed least significant bits that are chosen by default. Iteratively increasing the value of the global selects the next adjacent set of bits.

seq_level3_solver_granularity

Controls the granularity of problems passed to the level 3 engine. Legal values are wordwise and all_together. Default: wordwise.

Description

The seq_level3_solver_granularity global controls the granularity of problems passed to the level 3 engine. By default, the engine is called once for each word-level problem; all bits associated with the word-level problem are passed simultaneously.

Specify all_together to pass all word-level problems to the engine simultaneously. This is useful when all word-level problems are related or share similar structure. However, there is a risk of overwhelming the solvers with too much logic or adversely impacting caching.

seq_perform_state_checks

Determines the type of state check to be performed. Legal values are: all, only_bec, and none. Default: all.

Description

The global seq_perform_state_checks determines the type of state check to be performed. SLEC performs two types of checks:

1. On a transaction-to-transaction basis, identify flops that are equal to their reset values.
2. At the end of each Bounded Equivalence Check(BEC) transaction, identify flops that have gone back to their reset values.

To perform both types of check, set the seq_perform_state_checks global to all.

To perform the second type of check only, set the seq_perform_state_checks global to only_bec.

To perform neither of the checks, set the the seq_perform_state_checks global to none. In this case, the full proof mode will be disabled.

Tip
Use the following to guide your setting for this global variable:

- If too much runtime is being spent in identifying the inductive state variables, set this global to only_bec.
- After posing all solver problems, if SLEC goes into a hang while performing checks for BEC transactions, it is recommended that this global be set to none.

set_zero_replicate_to_null

Determines how SLEC interprets multi-concatenated expressions with non-positive repeat values. When this global is enabled, such expressions are set to NULL. Legal settings are 0 and 1. Default setting is 1.

Description

When this global is set to 1, multi-concatenated expressions with non-positive repeat values are replaced with NULL. When this global is set to 0, such expressions are replaced with 1'b0. Consider the following example.

```
assign c = { {0{1'b1}}, a[31:0] };
assign c = { {1'bx{1'b1}}, a[31:0] };
assign c = { {1'bz{1'b1}}, a[31:0] };
assign c = { {-1{1'b1}}, a[31:0] };
```

For the above mentioned example, when this global is set to 1, the expressions will be treated as:

```
assign c =  { a[31:0] };
```

When set to 0, the expressions will be treated as:

```
assign c = { 1'b0, a [31:0] }
```

show_all_sim_falsifications

By default, simulation-based validation only shows the earliest falsification for a map. Turning this global on enables to it show all subsequent falsifications (for as long as simulation runs) on such a map. This global also requires you to disable the stop_at_first_falsification global.

soft_runtime_limit

Sets overall time limit for SLEC

Description

If the overall run time exceeds the value specified by this global variable, SLEC will shut down gracefully. This value is a string which typically includes an integer followed by an optional time unit. If no time unit is specified, then SLEC assumes the time unit is *seconds*. If "0" is specified, then SLEC implies unlimited run time. By default, this global variable is set to "0". If SLEC can not parse the string value, then it will issue an error.

Examples (in bold) of acceptable string format include:

- *seconds*: **45** or **45s**
- *minutes*: **30min** or **30m**
- *hours*: **7hours** or **7h**
- *days*: **2days** or **2d**
- *mixed*: **1:20:10** implies 1 hour 20 minutes and 10 seconds

sim_based_validation

Enables the simulation engine to find easy counterexamples for the equivalence checking problem. Legal settings are 0 or 1. Default setting is 1.

Description

Sometimes simulation may find a false negative (mismatches not interesting to the user) due to conservative 3-valued simulation. To proceed further, this switch can be set to 0, forcing the formal solvers to prove or disprove the equivalence. The mismatch found by the solver (if any) will likely be a shorter counterexample and easier to investigate.

sim_dump_aux_checkers

Boolean variable that specifies whether to include auxiliary checker signals in waveform dumps. Default: false.

Description

This global has an effect only when -auxsignals is specified with the config_trace_files command.

sim_finds_only_earliest_mismatches

Specifies whether or not simulation should stop at the first transaction with mismatching maps. Legal settings are 0 or 1. Default setting is 1.

Description

By default, during simulation-based validation, SLEC will stop at the first transaction exhibiting map mismatches and report all mismatches found so far. If sim_finds_only_earliest_mismatches is set to 0, then simulation will be performed for the number of transactions specified by the global sim_max_transactions and all mismatching maps found are reported. Another side-effect of turning off sim_finds_only_earliest_mismatches is that if waveforms are being dumped for simulation-based validation, that is, "config_trace_files -simdump" has been specified, then these waveforms will be for the length of sim_max_transactions even if a falsification has been found earlier.

sim_max_transactions

Specifies the number of transactions for simulation based validation. Legal setting is any positive integer. Default setting is 100.

Description

Tune to see more/less transactions in simulation-based verification waveforms. The default is 100.

solver_cache_disk_usage

Defines the size of disk to be allocated for solver caching. Default: 100 Mb.

Description

Use the solver_cache_disk_usage global to specify the space to be allocated for solver caching.

solver_cache_location

Defines the location of the cache, where the path provided should start with a forward slash (/). Other legal settings include homedir, workdir and none. Default: homedir.

Description

Use the solver_cache_location global to specify the path to be used for solver caching. If providing a specific path, make sure the path starts with a forward slash (/). Other acceptable values include:

- homedir: Points to the *~/.slec/solver_cache/* directory. This is also the default solver cache location.
- workdir: Points to the *<current working directory>/.solver_cache/* directory.
- none: Disables solver caching. Instead, all calls go directly to the solver.

solverloop_num_processes

This will set the upper limit for the maximum number of SLEC Worker licenses requirement for enabling parallelization. Parallelization will be enabled only when this global's value is more than 1. Default value is 0.

spec_output_latency

Specifies the latency of the specification design for name-based output maps. Legal setting is any non-negative integer. Default setting is 0.

Description

If the specification design is pipelined, it starts producing meaningful outputs after the specified number of cycles. An output is termed meaningful if the user cares to check it against the corresponding output of the other design. By default it is assumed that a design produces meaningful outputs starting first cycle. The default can be overridden by using this setting. This value is only used for output ports or blackboxed input ports implicitly mapped by name.

For cycle-accurate designs it is neither necessary nor desirable to specify the real output latency. If the specification and implementation designs can be compared every cycle, then this global should be allowed to default to 0. For optimum design capacity, the smallest possible value for output latency should be used.

Note that this global does not affect flop maps or explicit output maps.

spec_throughput

Specifies the throughput of the specification design, in cycles. Design throughput of <*n*> cycles indicates that the design consumes inputs and produces meaningful outputs every <*n*> cycles. The default throughput is 1. Legal setting is any positive integer. To specify the throughput of the implementation design, use the global impl_throughput.

Description

The globals spec_throughput and impl_throughput specify the number of cycles required by the specification and implementation designs to transition to a state of equivalence where they:

- Consume mapped inputs (provided by SLEC)
- Produce mapped flops or outputs that need to be verified by SLEC

Consider a typical scenario where an untimed C++ model is compared with its RTL implementation. Assuming the C++ model consumes inputs every cycle and produces an output immediately. The global spec_throughput would be set as follows:

```
set_global spec_throughput 1
```

Now, assuming the corresponding RTL takes 200 cycles to produce equivalent output and is ready to accept inputs corresponding to the second cycle of the C++ model in the 201st cycle, the global impl_throughput would be set as follows:

```
set_global impl_throughput 200
```

For cycle-accurate designs it is neither necessary nor desirable to specify the real throughput. If the specification and implementation designs can be compared every cycle, then this global should be allowed to default to 1. For optimum design capacity, the smallest possible value for throughput should be used.

For example, the designs being compared could be a RTL block which implements an image-processing transform and another cycle-accurate version of it. From the designer's perspective, the throughput could be several thousands of cycles that the block takes to complete the transform. However, given that the designs are cycle-accurate, they can be compared with both the throughputs set to 1.

stop_after_concolic_engine

Boolean variable that specifies whether to stop the verification flow after concolic verification phase. This global is ignored if enable_concolic_engine is set to false. Default: false.

stop_after_low_effort_engines

Boolean variable that specifies whether to stop the verification flow after low-effort verification engines (LL3 and LL1).

stop_at_first_falsification

Instructs SLEC to stop proving the remaining maps as soon as it encounters a falsification. Chances of getting a conditional proof increase when this global is enabled. Legal settings are 0 or 1. Default setting is 1 so SLEC stops as soon as it encounters a falsification.

Description

Set this global to 0 to let SLEC continue proving remaining maps even when it has found falsifications. Setting this global to 0 allows one to sweep through the full set of output maps. Note that falsifications found for initial value checks for flop maps and inductive maps still result in SLEC stopping after all initial value checks have been performed, as the initial value checks are required for soundness of induction.

Note:
When this global is enabled, testbenches are disabled.

system_verilog_version

Specifies the default SystemVerilog version for interpreting SystemVerilog design files. Legal values are: 2005, 2009, 2012, 2017. Default: 2017.

Description

This global sets the default SystemVerilog version for the build_design, read_design, and read_sv commands.

systemc_version

Sets the version of the SystemC header files to be included during compilation of a SystemC design. Legal settings are 2.2 and 2.3. Default: 2.2.

Description

By default, SLEC includes SystemC 2.2 header files when compiling a SystemC design. To include the SystemC 2.3 version of the header files, set this global to 2.3 before executing the build_design command.

testbenches_dump_waveforms

Controls waveform dumping from counterexample testbenches. Legal settings are 0 or 1. Default setting is 1.

Description

If the global is set, SLEC will generate code within the counterexample testbenches such that waveforms (for example, VCD files) are dumped as simulation progresses.

testbenches_systemc_binary_type

Controls the binary type of the object compiled by the SystemC testbench makefile. Legal values are 32bit and 64bit. The default is 32bit.

Description

This global ensures the compiled object binary type matches the OSCI installation settings. If set to 32bit, GCC is invoked with the -m32 option to force 32-bit binaries. If set to 64bit, GCC is invoked with -m64 (only on x86_64 platforms).

top_design_view_scope

Determines the name of the top level module that instantiates the spec and impl designs for deisign/ waveform viewing.

unmap_implicit_x

Specifies not to consider implicit X's during structural matching of X's (enabled with the global map_x_as_symbolic).

Description

If this global is set to 1, implicit X's are not considered for structural matching. By default, all X's are considered for matching. Legal values are 0 and 1. Default setting is 0.

unmap_implicit_z

Specifies not to consider implicit Z's during structural matching of Z's (enabled with the global map_z_as_symbolic).

Description

If this global is set to 1, implicit Z's are not considered for structural matching. By default, all Z's are considered for matching. Legal values are 0 and 1. Default setting is 0.

use_relative_paths

Specifies that SLEC use relative pathnames while referring to the tool generated files. Legal settings are 0 or 1. Default setting is 1.

Description

If the global is not set, SLEC will use absolute pathnames while referring to the tool-generated log files, waveforms, testbenches, and so on.

value_on_input_pins_during_reset

Sets undriven input ports to X, 0, or 1 during reset. Legal settings are X, 0, or 1. Default setting is X.

Description

By default, during reset, any unconstrained primary input ports or black box output ports are driven by X values. This value may be changed to be all 0 or 1. Constraining these ports to constant values may minimize the amount of X states present in the designs after reset.

verilog_version

Specifies the default Verilog version for interpreting Verilog design files. Legal values are: 1995, 2001, 2005. Default: 2005.

Description

This global sets the default Verilog version for the build_design, read_design, and read_verilog commands.

vhdl_version

Specifies the default VHDL version for interpreting VHDL design files. Legal values are: 1987, 1993, 2000, 2002, 2008. Default: 2008.

Description

This global sets the default VHDL version for the build_design, read_design, and read_vhdl commands.

warn_for_undriven_signals

Issues warning on signals that are not driven by primary inputs, black box outputs, or other logic. Legal settings are 0 or 1. Default setting is 0.

Description

If the global is set, SLEC will issue warning signal on undriven signals: "[CPT-DCUE] (/home/qa/temp/t.v:20) Signal 't' in module 'top' is undriven." This global does not control the warning issued on ports of modules that are undriven ([CPT-DC6]).

Chapter 4
Command Line Programs

This chapter describes the command line tools for invoking SLEC and generating testcase packages.

You can invoke SLEC to either run a Tcl script, which contains all instructions for performing the desired verification tasks, or to launch an interactive command shell. The interactive comand shell interprets Tcl commands to control problem setup, verification, and results reporting.

In some cases, you may need to port the entire testcase to another computer for debug. The package_testcase tool allows you to package all the required files for reproducing the testcase behavior on another computer. This may be especially useful when technical support is required.

package_testcase

Creates a SLEC testcase package for porting to another computer for debugging purposes.

Usage

package_testcase **--workdir *dir*** [--no-tcl] [--no-design-files]

Arguments

- **--workdir *dir***

  Specifies the work directory to capture in the package. You can specify this argument multiple times to capture multiple work directories.

- --no-tcl

  By default, the command includes a snapshot of the TCL setup code. Specify this argument to omit the TCL code from the package.

  Note:
  Specifying this argument can make it more difficult to reproduce the testcase.

- --no-design-files

  By default, the command includes a snapshot of design files for the testcase. Specify this argument to omit the design files from the package.

  Note:
  Specifying this argument can make it more difficult to reproduce the testcase.

Description

The package_testcase tool creates a tar package of all the required files to reproduce a SLEC result on another computer. The tool generates an output file with the following name format: *slec-testpkg-YYMMDD.tar.gz*. If such a file already exists, it appends an index number. If you do not include any arguments, package_testcase simply prints usage information and exits.

To effectively reproduce the testcase, provide a work directory where SLEC built the design. If you provide a work directory where SLEC simply reloaded a model database with the read_db command, the resulting package will be insufficient for reproducing the testcase.

After the tool exits, inspect the contents of the resulting tar package to ensure all required files are included.

slec command

Use the **slec** command to start the SLEC tool.

Usage

slec [*options*] [*tcl-script*] [*other-arguments*]

Arguments

- *tcl-script*

  Name of a script file containing Tcl commands to execute within SLEC.
- -eval *command-string*

  Evaluate the string of Tcl commands at startup.
- -interactive

  Stay in the interactive shell even if the Tcl script calls exit.
- -help

  Show this message and exit.
- -nocopyright

  Suppress the verbose copyright preamble message at startup.
- -queuelic

  Wait indefinitely for license instead of erroring out.
- -queuelic_limit *value*

  Wait for license with a time limit, such as 30s or 1h.
- -version

  Show the tool version and exit.
- -workdir *directory-name*

  Change the location of the SLEC Work Directory so Catapult Formal writes the output to <*directory-name*> instead of the default *calypto*<*n*> directory.
- -mgls_license_file *string*

  Specify the location of the MGLS license file.

Description

The **slec** command with no parameters provides an interactive command shell. Otherwise, SLEC runs the TCL script, runs the command string, or issues a help message. Upon completing the script, SLEC normally exits. The -interactive option forces SLEC to remain in an interactive TCL shell where additional commands may be executed.

SLEC returns an error code of 0 for normal execution or non-zero for any other condition.

Using the command string option and the interactive option are useful when creating or debugging a TCL script. For example:

```
slec -eval "read_design -spec spec.v"
```

will read the modules in the spec.v verilog file into SLEC. Any errors reading the design will be written to standard out and reported in the log file. An example using the interactive option:

```
slec -interactive test.tcl
```

starts SLEC, executes the commands in the *test.tcl* script and remains in the interactive shell even if the script calls exit. Reporting and status commands can then be entered to explore the setup and verification results.

Chapter 5
Design Representation

This chapter describes how design descriptions are represented within SLEC.

Design Libraries

SLEC supports design descriptions written in Verilog, VHDL and SystemC. A behavioral subset of Verilog and VHDL is supported including the common RTL synthesizable subset. SystemC support includes a behavioral subset of the SystemC library and most C++ constructs.

Read designs into SLEC with the read_design command. SLEC loads the modules defined in these files into the appropriate libraries.

The following libraries are pre-defined:

- **Specification (spec)** : the specification design library
- **Implementation (impl)** : the implementation design library
- **Constraints (cons)** : a library holding additional design constraint modules

After all you have read all modules into a design library, you can link the design with the link_design command. SLEC issues an error if it cannot find a module in the library. Once you have linked the design, run the build_db command to build SLEC's internal database for the design.

Design Elements

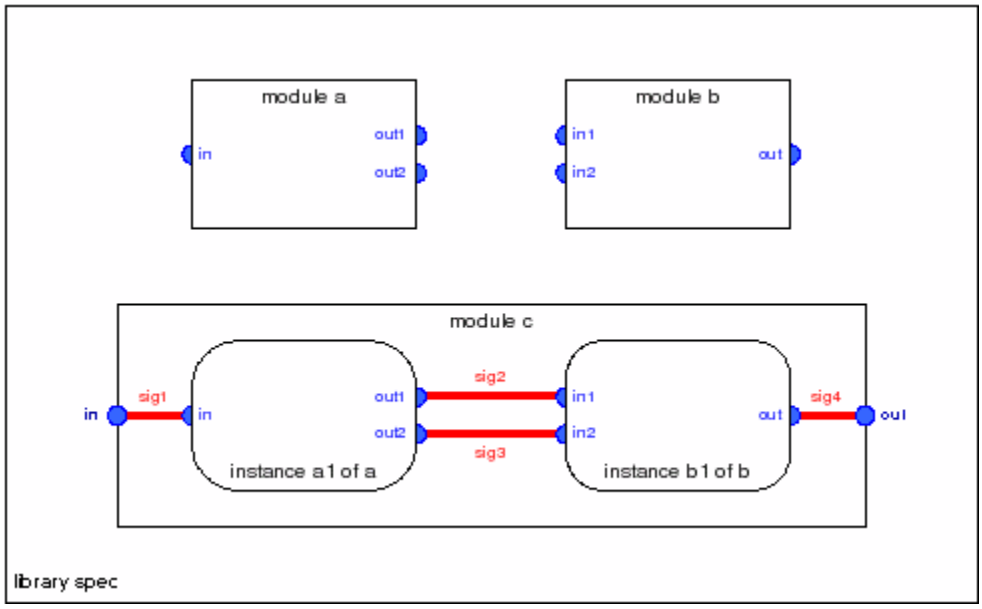
A <*module*> is the basic structural unit in SLEC. It corresponds to a module in Verilog, entity in VHDL or sc_module in SystemC. A module's interface is described by its <*port*> list, for example, a list of input and output ports in Verilog and VDL or public sc_in<T> and sc_out<T> ports in SystemC.

A module may contain instances of other modules. Ports of module instances, commonly also referred to as pins, are connected by signals. A signal connecting two module instance ports corresponds to a wire in Verilog, signal in VHDL or sc_signal<T> in SystemC.

SLEC creates instances of both designs' top modules. The instance of the top module of the specification design is referred to as the spec instance, and the instance of the implementation library is referred to as the impl instance.

Figure 1: Example Spec Library shows an example of a specification design library that includes three modules.

Figure 1. Example Spec Library



In the example spec library above, module c is the only module not instantiated in another module. SLEC therefore creates an instance 'spec' of module c as the top level specification module. Read, link, and build a specification design as follows:

```
build_design -spec design_modules.v other_modules.v
```

Setup commands use -spec or -impl options to refer to the specification or implementation designs respectively. In this example, SLEC reads in all the modules in the two design files. The files may contain Verilog or SystemC design descriptions. If a unique top level module is found in the design, it is assigned to be the top level module. Otherwise, the top level module may be explicitly assigned with the -top argument as follows:

```
build_design -spec -top specs_top_design_module design_modules.v
 other_modules.v
```

Each design's top level module has a set of I/O ports. Each port is an input signal port, an output signal port, or a clock port. Setup commands use spec. or impl. prefixes to refer to instances of the specification or implementation's top-level modules respectively. For example, you can access the clock port on the implementation's top level module with the signal name impl.clock.

Each input and output port has a type representing the type of information passed across the port and determined from the modules interface. Common types are bit, bit-vector, integer, or more abstract data types.

Top Modules

The specification and implementation designs must each have a corresponding top module defined. If a design has one module that is not instantiated in any other module, SLEC designates that module as the top module. If there are multiple top modules or you want to run verification on another module, specify the -top argument with the build_design or link_design command.

SLEC creates instances of both designs' top modules. The instance of the specification design's top module is named *spec*, and the instance of the implementation's top module named *impl*. For example, module c in Figure 1: Example Spec Library is the only module not instantiated in another module. Therefore, SLEC creates an instance 'spec' of module c as the top level module.

Black Box Modules

A black box module defines its interface ports but does not define its internal behavior.

A black box module is explicitly created with the create_black_box command. The internal logic of the module is removed, but the module interface is preserved and can be constrained during setup. Black box outputs become primary inputs of the design while black box inputs become primary outputs of the design. Either one or all instances of a module can be blackboxed, for example:

```
create_black_box -impl -module mem_bank
```

All instances of the mem_bank module are blackboxed in the implementation design.

Clocks

Ideal clocks, external to the designs, are defined and used to drive the clocking networks of both designs. Every port and signal in a design is associated with an ideal clock. Clock ports will be directly associated with ideal clocks. Non-clock ports and signals will be sensitive to edges of ideal clocks. Refer to the create_clock, set_clock_root, and set_clock_domain commands for additional information.

Naming Operators

To refer to design elements in the Tcl setup file, modules, instances, signals, and ports are named using the same language syntax as was used in the file which defined them. The following naming conventions are used to refer to design elements in Verilog, VHDL and SystemC.

Table 3. Operators by Language

Operator	Verilog	SystemC	VHDL
Hierarchical instance divider	.	.	.
Port or signal reference	.	.	.
Bit-select	[ ]	[ ]	( )
Part-select	[:]	[:]	(to) and (downto)
Array indexing	[ ]	[ ]	( )
Multidimensional array	not supported	[ ] [ ]	( ) ( )
Member reference	n/a	.	.
Pointer to member reference	n/a	0>	n/a

Table 3. Operators by Language (continued)

Operator	Verilog	SystemC	VHDL
Superclass member scope	n/a	*<class-name>*::	n/a

Because brackets "[...]" and spaces have a special meaning in Tcl, use of brackets for bit select, part select, or array indexing should be prefixed with a backslash "\[...\]" or "(2\ downto\ 0)" to avoid misinterpretation. Alternatively, braces can be used to delineate the entire statement: "{...[...]}" and "{..(2 downto 0)}". In this case, the escape character (\) is not required.

Note:
If the contents of [ ] start with a number, SLEC will not interpret it as a proc call as Tcl does. This feature allows you to index arrays without needing to use the escape character every time. For example, you could issue the following Tcl command without causing an error:

```
[create_waveform -sample_start 2 impl.m1.data_q[4:2]]
```

Instances of modules within a design may be specified by a hierarchical path starting from their top instance. Using the example spec library of Figure 1: Example Spec Library on page 232, the instance of module B within the instance of top-module C would be referred to as spec.b1. The output port of instance b1 would be referred to as spec.b1.out.

In SystemC, members of a class may be accessed directly using the dot (.) operator or indirectly through the pointer operator (->). Additionally, members which are inherited from a superclass should include the superclass scope in their name. In the previous example, if instance b1 's out port had an integer field count, it would be referred to as spec.b1.out.count. If the count field had been inherited from a superclass S, then the count field would be referred to as spec.b1.out.S::count instead.

Appendix A
Pattern Matching Syntax

Pattern matching is used to look up named identifiers in the design database and for specifying file names and paths. Both regex and glob styles are supported.

The regex style is the Perl-compatible regular expression style, which allows control over literals, character classes and sets, wildcards, repeats, grouping, alternatives, anchors, and escape sequences.

The glob style is the UNIX-style wildcard pattern, a simpler form of expression which allows only wildcards and character ranges.

Portions of pattern matching functionality are derived from the boost.org regular expression library; portions of the documentation within this appendix are derived from the Boost.regex documentation, copyright John Maddock 1998-2003.

Regular Expression Syntax

This section describes the supported set.

Literals

All characters are literals except: ., |, *, ?, +, (,), {, }, [,], ^, $ and \. These characters are literals when preceded by a \.

Glob

The dot character . matches any single character except newline character.

Repeats

A repeat is an expression that is repeated an arbitrary number of times. An expression followed by * can be repeated any number of times including zero. An expression followed by + can be repeated any number of times, but at least once. An expression followed by ? may be repeated zero or one times only. When it is necessary to specify the minimum and maximum number of repeats explicitly, the bounds operator "{}" may be used, thus "a{2}" is the letter a repeated exactly twice, "a{2,4}" represents the letter a repeated between 2 and 4 times, and "a{2,}" represents the letter a repeated at least twice with no upper limit. Note that there must be no whitespace inside the {}, and there is no upper limit on the values of the lower and upper bounds. All repeat expressions refer to the shortest possible previous subexpression: a single character; a character set, or a subexpression grouped with "()" for example.

Examples

"ba*" will match all of "b", "ba", "baaa" etc.

"ba+" will match "ba" or "baaaa" for example but not "b".

Nongreedy Repeats

Nongreedy repeats are possible by appending a "?" after the repeat; a nongreedy repeat is one that will match the shortest possible string.

Parentheses

Parentheses serve two purposes: to group items together into a subexpression, and to mark what generated the match. For example the expression "(ab)*" would match all of the string "ababab". It is permissible for subexpressions to match null strings. If a subexpression takes no part in a match (for example if it is part of an alternative that is not taken) then both of the iterators that are returned for that subexpression point to the end of the input string, and the matched parameter for that subexpression is false. Subexpressions are indexed from left to right starting from 1, subexpression 0 is the whole expression.

Nonmarking Parentheses

Sometimes you need to group subexpressions with parentheses, but do not want the parentheses to emit another marked subexpression. In this case, nonmarking parentheses (?:<*expression*>) can be used. For example the following expression creates no subexpressions:

```
"(?:abc)*"
```

Forward Lookahead Asserts

There are two forms of these; one for positive forward lookahead asserts, and one for negative lookahead asserts:

"(?=abc)" matches zero characters only if they are followed by the expression "abc".

"(?!abc)" matches zero characters only if they are not followed by the expression "abc".

Alternatives

Alternatives occur when the expression can match either one subexpression or another, each alternative is separated by a |. Each alternative is the largest possible previous subexpression; this is the opposite behavior from repetition operators.

Examples

"a(b|c)" could match "ab" or "ac".

"abc|def" could match "abc" or "def".

Sets

A set is a set of characters that can match any single character that is a member of the set. Sets are delimited by [and] and can contain literals, character ranges, character classes, collating elements and equivalence classes. Set declarations that start with ^ contain the compliment of the elements that follow.

Examples

Character literals:

"[abc]" will match either of a, b, or c.

"[^abc]" will match any character other than a, b, or c.

Character ranges:

"[az]" will match any character in the range a to z.

"[^AZ]" will match any character other than those in the range A to Z.

Note that character ranges are highly locale dependent: they match any character that collates between the endpoints of the range; ranges will only behave according to ASCII rules when the default "C" locale is in effect.

Character classes are denoted using the syntax "[:<*classname*>:]" within a set declaration, for example "[[:space:]]" is the set of all whitespace characters. The available character classes are:

alnum	Any alphanumeric character.
alpha	Any alphabetical character a-z and A-Z. Other characters may also be included depending upon the locale.
blank	Any blank character, either a space or a tab.
cntrl	Any control character.
digit	Any digit 0-9.
graph	Any graphical character.
lower	Any lower-case character a-z. Other characters may also be included depending upon the locale.
print	Any printable character.
punct	Any punctuation character.
space	Any whitespace character.
upper	Any upper-case character A-Z. Other characters may also be included depending upon the locale.
xdigit	Any hexadecimal digit character, 0-9, a-f and A-F.
word	Any word character all alphanumeric characters plus the underscore.
unicode	Any character whose code is greater than 255; this applies to the wide character traits classes only.

There are some shortcuts that can be used in place of the character classes:

- \w in place of [:word:]
- \s in place of [:space:]
- \d in place of [:digit:]

- \l in place of [:lower:]
- \u in place of [:upper:]

Collating elements take the general form [.tagname.] inside a set declaration, where tagname is either a single character, or a name of a collating element, for example [[.a.]] is equivalent to [a], and [[.comma.]] is equivalent to [,]. The library supports all the standard POSIX collating element names, and in addition the following digraphs: "ae", "ch", "ll", "ss", 'nj", "dz", and "lj", each in lower, upper and title case variations. Multicharacter collating elements can result in the set matching more than one character, for example [[.ae.]] would match two characters, but note that [^[.ae.]] would only match one character.

Line Anchors

An anchor is something that matches the null string at the start or end of a line: ^ matches the null string at the start of a line, $ matches the null string at the end of a line.

Back References

A back reference is a reference to a previous subexpression that has already been matched, the reference is to what the subexpression matched, not to the expression itself. A back reference consists of the escape character \ followed by a digit 1 to 9, "\1" refers to the first subexpression, "\2" to the second etc. For example the expression "(.*)\1" matches any string that is repeated about its midpoint for example "abcabc" or "xyzxyz". A back reference to a subexpression that did not participate in any match, matches the null string.

Characters by Code

This is an extension to the algorithm that is not available in other libraries, it consists of the escape character followed by the digit 0 followed by the octal character code. For example, "\023" represents the character whose octal code is 23. Where ambiguity could occur use parentheses to break the expression up: "\0103" represents the character whose code is 103, "(\010)3" represents the character 10 followed by 3. To match characters by their hexadecimal code, use \x followed by a string of hexadecimal digits, optionally enclosed inside {}, for example \xf0 or \x{aff}, notice the latter example is a Unicode character.

Word Operators

"\w" matches any single character that is a member of the "word" character class, this is identical to the expression "[[:word:]]".

"\W" matches any single character that is not a member of the "word" character class, this is identical to the expression "[^[:word:]]".

"\<" matches the null string at the start of a word.

"\>" matches the null string at the end of the word.

"\b" matches the null string at either the start or the end of a word.

"\B" matches a null string within a word.

Buffer Operators

"\`" matches the start of a buffer.

"\A" matches the start of the buffer.

"\'" matches the end of a buffer.

"\z" matches the end of a buffer.

"\Z" matches the end of a buffer, or possibly one or more new line characters followed by the end of the buffer.

A buffer is considered to consist of the whole sequence passed to the matching algorithms.

Escape Operator

The escape character \ has several meanings. Inside a set declaration the escape character is a normal character. The escape operator may introduce an operator for example: back references, or a word operator. The escape operator may make the following character normal, for example "*" represents a literal * rather than the repeat operator.

Single Character Escape Sequences

The following escape sequences are aliases for single characters:

Escape sequence	Character code	Meaning
\a	0x07	Bell character.
\f	0x0C	Form feed.
\n	0x0A	Newline character.
\r	0x0D	Carriage return.
\t	0x09	Tab character.
\v	0x0B	Vertical tab.
\e	0x1B	ASCII Escape character.
\0dd	0dd	An octal character code,where dd is one or more octal digits.
\xXX	0xXX	A hexadecimal character code, where XX is one or more hexadecimal digits.
\x{XX}	0xXX	A hexadecimal character code, where XX is one or more hexadecimal digits, optionally a unicode character.
\cZ	z-@	An ASCII escape sequence control-Z, where Z is any ASCII character greater than or equal to the character code for '@'.

Miscellaneous Escape Sequences

The following are provided mostly for perl compatibility, but note that there are some differences in the meanings of \l \L \u and \U:

\w	Equivalent to [[:word:]].
\W	Equivalent to [^[:word:]].

\s	Equivalent to [[:space:]].
\S	Equivalent to [^[:space:]].
\d	Equivalent to [[:digit:]].
\D	Equivalent to [^[:digit:]].
\l	Equivalent to [[:lower:]].
\L	Equivalent to [^[:lower:]].
\u	Equivalent to [[:upper:]].
\U	Equivalent to [^[:upper:]].
\C	Any single character, equivalent to '.'.
\X	Match any Unicode combining character sequence, for example "a\x 0301" (a letter a with an acute).
\Q	The begin quote operator, everything that follows is treated as a literal character until a \E end quote operator is found.
\E	The end quote operator, terminates a sequence begun with \Q.

Wildcard Syntax

The "wildcard" style is much simpler but less powerful than the "regex" style. It offers the enduser a simpler way to express common match patterns. It permits the following syntax.

Asterisk (*)

The wildcard * character is equivalent to regex ".*", matching zero or more nongreedy occurrences of any literal.

Anycharacter (?)

The wildcard ? character is equivalent to regex ., matching precisely one literal.

Double Asterisks (**)

When searching across design instances, the double asterisks ** operator is made available with the glob style.

The intent of ** is to match zero or more layers of hierarchy. To match one or more layers of hierarchy, use *.**. A ** is only valid at the beginning of an expression or immediately after a '.'. A '.' must immediately follow **. A search expression may not end with **. A search expression may contain multiple ** and * operators.

For example, consider the following design instance hierarchy:

```
instance i
  ports p pp
  instance i
    ports p pp
  instance ii
    ports p pp
instance ii
  ports p pp
  instance i
    ports p pp
  instance ii
    ports p pp
```

The following table shows some search strings and resulting matches:

**.p	i.p i.i.p i.ii.p ii.p ii.i.p ii.ii.p
ii.**.p	ii.p ii.i.p ii.ii.p
i.**.p	i.p i.i.p i.ii.p
ii*.**.p	ii.p ii.i.p ii.ii.p
.i..p	i.p i.i.p i.ii.p ii.i.p
i.**.p*	i.p i.i.p i.ii.p i.pp i.i.pp i.ii.pp
***	error (perhaps **.* was meant)
i.**	error (perhaps i.**.* was meant)
i.i**p	error (perhaps i.i*.**.*p was meant)

Backreference and Implicit Subexpressions

The wildcard style does not provide a syntax for explicitly grouping matched subexpressions for backreferencing, the wildcard ?, *, and ** characters shall be interpreted as implicit subexpression groups.

Substitution Syntax

Substitution makes sense only in the presence of an accompanying pattern match. Each matching subexpression extracted from a pattern match is used towards the generation of a new string from a formatted string. Expansions are position based, ordered from left to right, counting from 1 up.

%N	Expands to the text that matched N'th subexpression
%0	Expands to all of the current match

%&	Expands to all of the current match

Third-Party Information

Details on open source and third-party software that may be included with this product are available in the *slec/legal* directory.