Static Design Analyzer (SDA)™

Version 3.6.22

User Guide

Surya Technologies, Inc.
4888 NW Bethany Blvd.
Suite K5, #191
Portland, OR 97229
USA

Phone: 503-260-9861
Email: customer_service@suryatech.com
Web: www.suryatech.com

Copyright Notice and Proprietary Information

Warranties

Surya Technologies, Inc., makes no warranty of any kind with regard to this material, except as set forth explicitly in a software license agreement.

Trademarks

Static Design Analyzer, *SDA*, *TimingCheck, DesignCheck* and *SignalCheck* are trademarks of Surya Technologies, Inc. Other trademarks are properties of the respective trademark owners.

TABLE OF CONTENTS

1. SDA USAGE

1.1 Introduction

The *Static Design Analyzer (SDA)* tool suite represents a new generation of *static design analysis tools*. The increasing size and complexity of VLSI designs, coupled with shorter design cycles, require new design verification methodologies that can exhaustively verify large designs in an acceptable time period. Static analysis techniques are needed to solve this problem, since simulation-based approaches cannot provide full coverage in any reasonable amount of time.

Static analysis techniques can be applied to a wide range of problems in VLSI designs. SDA currently operates at the gate level. It includes a complete static timing analysis and verification engine (referred to as *TimingCheck*), and a static design constraint verification engine (referred to as *DesignCheck*). SDA provides a set of powerful and integrated capabilities to statically analyze timing, signal integrity, electrical properties, functionality and topology in a single framework.

The gate-level timing analysis/verification uses either SDF delays or DSPF parasitics, and is enabled by the path reporting options described in Section 2. The timing analysis engine within SDA handles multiple clocks (multi-phase/frequency), transparent latches, user-specified false-paths, and user-specified multi-cycle paths. At present, sorted/ordered critical path reports are generated for the entire design. In addition, slack reports for specified nets and limited exhaustive path reporting for specified end-points are available. The user can specify whether worst-case or best-case analysis is desired.

The *DesignCheck* option of SDA is a static design constraint verification (DCV) tool that can exhaustively verify user-programmed rules and constraints in complex designs. *DesignCheck*'s verification complements existing static analysis tools and is targeted towards portions of the design verification problem that are typically not covered by other tools.

DesignCheck allows designers to express these static rules in a simple language and then checks the design exhaustively to detect all violations of these rules. The rules can cover a wide range of areas, such as structural path rules, cell usage rules, logical or functional rules and electrical rules. Together, these rules can be used to verify constraints or requirements in various domains, such as timing, power, signal-integrity, testability, cell usage, connectivity, etc.

Internally, *DesignCheck* can perform three types of analyses: timing analysis, functional analysis and topology analysis. The uniqueness of *DesignCheck* is that all these analyses can be performed within the same tool in a single framework, which allows analysis of multiple problem domains as well as interactions between these domains.

1.2 Command Line Options

The current version of SDA operates at the gate level. The following inputs are *required* to run SDA:

- Verilog or VHDL gate-level (cell-level) netlist file(s)
- Cell library file(s) containing both functional and timing characteristics of all leaf-level cells
- User-created command file containing constraint and setup information

One of the following *optional* inputs is used whenever timing information is required for verifying certain classes of rules:

- Standard Delay Format (SDF) file(s) containing pre-computed delay values for all cell instances and interconnects.
- Detailed Standard Parasitic Format (DSPF) file(s) containing extracted parasitic data for interconnects.

The command line usage of the tool is as follows.

```
Usage: sda [ options ]

    options:
     -nver <verilogNetlistFileName>
     -nvhdl<vhdlNetlistFileName>
     -lib <libFileName>
     -libmin <libMinFileName>
     -sdf <sdfFileName>
     -sdfmin <sdfMinFileName>
     -sdfmax <sdfMaxFileName>
     -sdfmax2 <sdfMax2FileName>
     -dspf <dspfFileName>
     -cmd <cmdFileName>
     -top <topModuleName>
     -out <outputReportFile>
     -log <outputLogFile>
```


The "-top" option can be used to specify the top-level module, *or any module defined in the netlist files*, as the highest level of the hierarchy for each run of SDA. Each input file option can be used multiple times to read in multiple files of each type.

1.3 Structural Netlist Inputs

The netlist files are simple hierarchical Verilog or VHDL structural (gate-level) netlists, where the leaf-level elements are instances of cells defined in a library. Limited subsets of the Verilog and VHDL languages[1] are supported, which are generally compatible with the netlist outputs of logic synthesis tools. One or more netlist files can be provided as input.

1.4 SDF Input

Delay and constraint back-annotation information in the Standard Delay Format (SDF), for both cells and interconnects, is supported. The constructs supported are compatible with Version 3.0 of SDF (as published by Open Verilog International). The specific SDF constructs and features supported are as follows:

(1) CELL, CELLTYPE and INSTANCE sections.
(2) DELAY section, including the following statements: IOPATH, PORT and INTERCONNECT. Both ABSOLUTE and INCREMENT delays are supported.
(3) TIMINGCHECK section, including the following statements: SETUP, HOLD, SETUPHOLD, RECOVERY, REMOVAL, RECREMOVAL, SKEW, WIDTH, and NOCHANGE.
(4) TIMINGENV section, including the following statements: PATHCONSTRAINT, SUM, LESS, DIFF, and SKEWCONSTRAINT.

The MIN and MAX values from each SDF statement will be read in and used. The TYP value will not be used. At present, most of the TIMINGCHECK and TIMINGENV statements require a single constraint value, except that SETUP and HOLD statements require the MIN, TYP and MAX values.

[1] The Verilog and VHDL language support is targeted at fully synthesized and elaborated designs. Behavioral or register-transfer level descriptions are not supported. Parameter definitions and usage are supported in Verilog only within a module, and can not be modified at any module instance. Other Verilog constructs such as "includes", "defines", built-in gate-level primitives and user-defined primitives are not supported. VHDL support is limited as well, and closely reflects the Verilog support.

The wild-card character (“*”) is generally not permitted within SDF delay annotation or timing constraint specification, except that a single ‘*’ can be used to replace the entire instance name field of any cell type. Escaped name strings are treated as per SDF rules (see the description for the command “%set_new_escaped_name_rules” in Section 10 for more details).

1.5 DSPF Input

The DSPF (Detailed Standard Parasitic Format) files can be used to back-annotate parasitic RC tree networks for interconnects. If there is no SDF annotation for a cell instance and its output interconnect net, then SDA will perform its own internal delay calculation. If DSPF parasitic data is available for that net, then detailed RC delay calculation will be performed internally.

Special notes on coupling capacitors:

Capacitors in the DSPF data can either be grounded or represent coupling capacitors between signal nets. Coupling capacitors between any two signal nets may be represented at both nets (preferred) or only represented at one of the two nets. Coupling can be directly represented within distributed RC networks.

Special notes on hierarchical DSPF:

DSPF files can be either flat or hierarchical – either one will work with hierarchical Verilog files. DSPF supports hierarchy by defining modules as "subcircuits", and then instantiating them. At each module instance, the net connections to the module pins must follow the "instance pin name" convention, where the hierarchical instance name of the module is followed by a delimiter (such as ":"), followed by the actual pin name.

DSPF does not define any specific syntax for specifying additional layout-dependent module ports to describe split pins and feed-through connections. The following specification describes an extended hierarchical DSPF supported by SDA.

Syntax for naming *additional* pins, when a single port is split into multiple ports:

<original_pin_name>&<string>

Each *additional* port of this group is named uniquely by appending the "&" character followed by a unique string, to the original port name. The first, or original, port name should not use this name extension. In addition, internal nodes of the RC network must use the original port or net name as the “base” name.

A new port can be added (for pure feed-throughs) by choosing any arbitrary pin name preceded by the "@" character. As before, *additional* ports in such a group can be uniquely named by appending the "&" delimiter followed by a unique string.

Multiple ports must be connected *within* the module through the RC network, if any module-level analysis is to be performed. If the analysis is only to be performed from an upper level, then it is sufficient to provide the connectivity between the multiple ports at the upper level (*outside* the module).

The standard DSPF "instance pin name" convention is also applicable to any new or additional *feed-through* pin.

General requirements:

- DSPF "subcircuits" (modules) must be defined first, before instantiation. Such a prior definition of a subcircuit can occur within a single DSPF file, or within another DSPF file that has already been read in. DSPF files are read by SDA in the same order that they are specified on the SDA command line.

- Each DSPF subcircuit must only have one instance in the design. If the same module or subcircuit is to be instantiated multiple times, then the module/subcircuit must be "uniquified" for each particular instance. This requirement is consistent with the fact that RC parasitics may be different within each instance.

- Hierarchical net/pin names within each subcircuit must be "relative" names, within the scope of the subcircuit only.

- Coupling from an internal net of a subcircuit can take two forms: (a) Coupling to another internal net of a subcircuit: relative hierarchical names must be used; (b) Coupling to a hierarchical pin anywhere in the design: full hierarchical name must be used.

- Feed-throughs can not be recursive, and can only penetrate one level of hierarchy.

- SDA supports a maximum hierarchical depth of 3 for DSPF netlists. This is consistent anyway with the fact that the physical hierarchy (reflected in the DSPF netlists) and the logical hierarchy (reflected in the Verilog netlists) may have a one-to-one correspondence only at the top few levels of the design hierarchy.

Special cases of DSPF back-annotation:

- Flat hierarchical names: When the Verilog netlist has not been flattened, but where the DSPF is flat, the net and cell instance names in the DSPF file(s) should be hierarchical name strings (corresponding to the Verilog hierarchy) without any escape characters.

- Escaped flat hierarchical names: When the Verilog netlist has been flattened and contains escaped name strings – i.e., flattened names corresponding to the original hierarchy, but with an escape character ("\") at the beginning signifying that all special characters (such as hierarchy dividers) lose their meaning until the next white space – flat DSPF can still be back-annotated without any changes. The flat DSPF file(s) can contain the escaped name strings for net names and cell instance names; however, true hierarchical DSPF cannot be used with a flattened Verilog netlist.

- Two-level DSPF hierarchy: A common case of hierarchical DSPF consists of two levels of hierarchy, with the DSPF subcircuits at the second level containing flattened net and cell instance names. The Verilog hierarchy is usually deeper than two levels. The net and cell instance names within the second level DSPF subcircuits should follow the rules stated above for flat DSPF (i.e., flat hierarchical names, or escaped flat hierarchical names). The ports of these second level DSPF subcircuits can use escaped name strings, except that the "&" character (for defining additional layout-dependent ports/pins) will still carry its special meaning within an escaped name. Port/pin names preceded by the "@" character should not be escaped.

- As an option, standard SDF rules (as opposed to Verilog rules) can be used to interpret escaped name strings in a more flexible way. Under this option, each escape character affects only the character that immediately follows it. (See the description for the command "%set_new_escaped_name_rules" in Section 10 for more details.)

1.6 Command Input

The command file allows the user to define timing attributes at primary inputs, outputs and clocks, static constraints to be verified, etc. It also allows the user to select reporting options and several other features supported by SDA.

Name strings:

Hierarchical cell instance names or net names are specified in Verilog style, with the "/" (slash) character acting as the hierarchy divider. A leading "/" character is required in all hierarchical names. Square brackets are permitted in name strings to refer to a *single bit* of a bus.

Hierarchical net or instance names which include a "\" (back-slash) character, indicating an "escaped" name string, are legal *only* if the back-slash occurs at the beginning of the last hierarchical level of the name string – any "/" (forward-slash) following the back-slash is illegal. This restriction is necessary to remove any ambiguity in the name strings.

As an option, standard SDF rules (as opposed to Verilog rules) can be used to interpret escaped name strings in a more flexible way. Under this option, each escape character affects only the character that immediately follows it. (See the description for the command "%set_new_escaped_name_rules" in Section 10 for more details.)

Name strings, such as cell names, instance names or net names, can be specified with one or more embedded wild-card characters '*' (asterisk characters). Each occurrence of a wild-card character represents a sequence of zero or more arbitrary characters. Wild-cards can be used with most command statements, with the following restrictions:

- Wild-card notation must be used in a reasonable and limited way, since the cost of searching the design for wild-carded names is simply additional runtime and memory requirements.

- In hierarchical instance or net names, each level of hierarchy must be explicitly separated with a "/" (forward-slash). The hierarchy divider can not be replaced with the wild-card character.

- A wild-card character '*' represents *one or more* actual characters in a string (and not zero or more), unless it occurs at the end of a string.

- If multiple wild-card characters are used within a string*, the sub-strings (which are the partial strings between successive wild-card characters) must be made unique* such that any particular sub-string is not found after the next wild-card character.

- Wild-cards are not permitted in the following cases:

  ⇒ Whenever net pairs, pin pairs or pin lists are specified, such as in cell_check, func_check and coupling_check commands.
  ⇒ Setting non-partitionable module instances.

Comments:

Each statement in the command file must start with one of the specified keywords described in the previous sections, along with additional arguments or

parameters, and can be written on single or multiple lines. SDA will ignore any comments starting with '/*' and ending with '*/', anywhere in the command file. At present, comments can not be started until after the *first* space, tab or newline character in the command file.

Note that the character sequence "/*" will be interpreted as the start of a comment – therefore, the wild-card character ("*") should not be inserted immediately following the "/" character in a hierarchical name string.

Miscellaneous:

- All time units in the command file are in pico-seconds.
- Multiple Specifications: If multiple commands attempt to set any attributes at a particular pin/net, the last command will prevail.

1.7 Outputs

Output files:

Reporting uses *stdout* as the default output. This can be changed to a user-specified output file using the *-out* option. Further, if multiple output files are selected through the command file, then various output files are generated. Each additional output file uses the name of the base output file, followed an extension. The results of timing analysis/verification and other static constraint verification are written to the base output file. SDF outputs are written to a file named *output_filename.sdf*; DSPF outputs are written to *output_filename.dspf;* spice decks are written to *output_filename.spice* or *output_filename.cone_i.spice*; and ECO (timing optimization) instructions are written to *output_filename.eco.size, output_filename.eco.buffer, and output_filename.v.*

Time units:

All time units in the output files are in pico-seconds.

Errors and warnings:

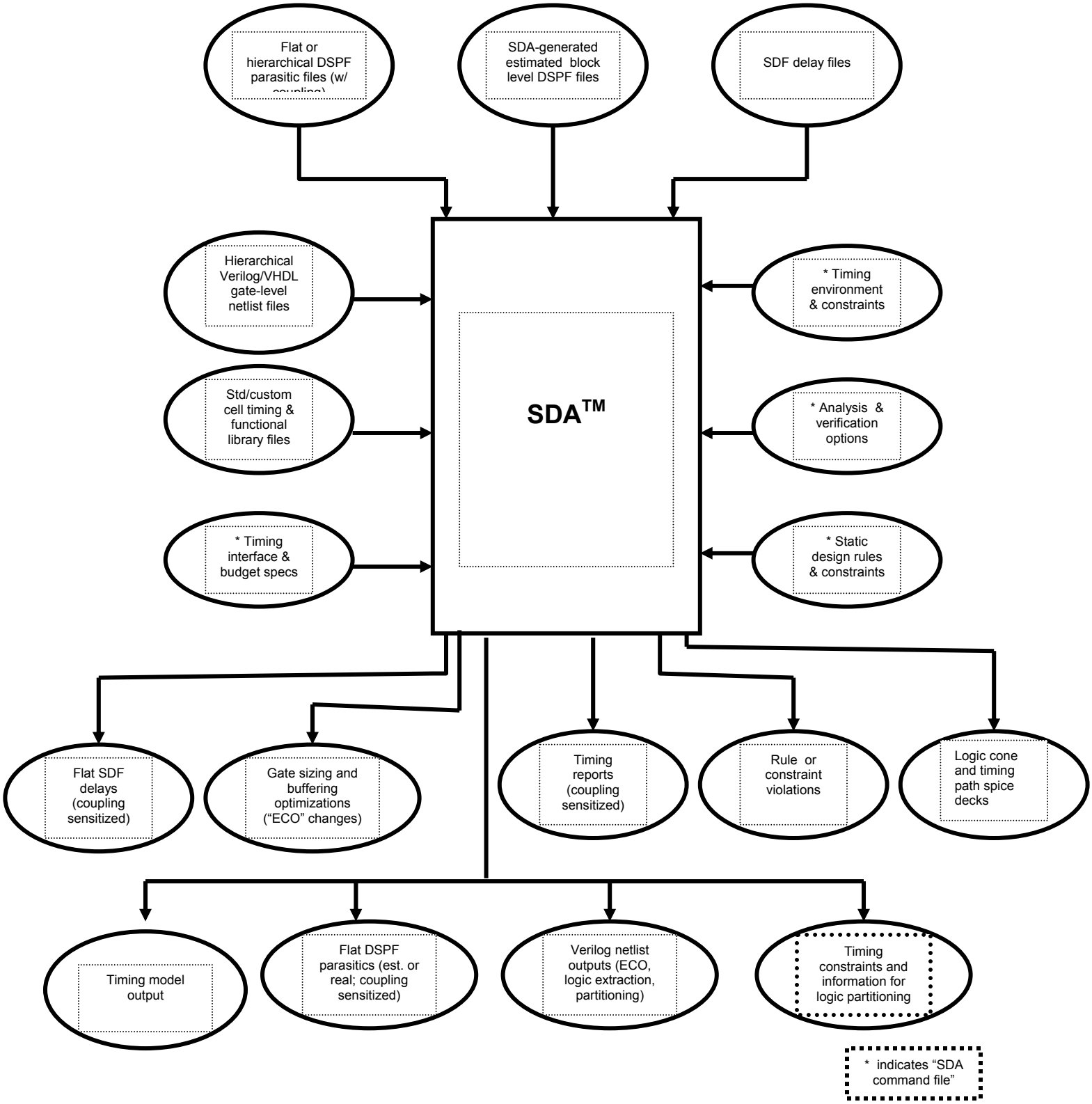
Errors, warnings and progress information are directed to *stderr*. This can be changed to a user-specified file using the *-log* option.

SDA outputs several types of errors and warnings during the course of an analysis. Each type of error or warning consists of a unique code, which is printed along with the message. Any syntax violation in an input file (missing or mis-spelled keyword, or missing arguments) will be reported as a fatal error.

Error messages are preceded by “<E>“ and warning messages are preceded by “<W>“. All errors are fatal, while most warnings are informational.

Each type of error or warning may apply to many similar cases, using the same error or warning code. All these messages are directed to stderr or to the file specified by the *-log* command line option.

Static Design Analysis/Optimization/Partitioning with SDA™
Inputs and Outputs



2. OVERVIEW OF SDA FEATURES

This section provides an overview of the features available in the *TimingCheck* and *DesignCheck* static analysis/verification engines within SDA.

2.1 Static Timing Analysis, Verification and Optimization (TimingCheck)

SDA's *TimingCheck* option offers a wide range of features to support accurate and efficient timing analysis of current and next generation designs.

TimingCheck offers the following significant features:

- Fast and efficient analysis of very large designs.
- Easy to set up and use.
- Handles multiple phase and multiple frequency clocks.
- Handles latch-based designs, including latch transparency issues.
- Includes a built-in delay calculator, which can compute accurate cell delays and interconnect delays.
- Reads interconnect parasitic RC data (in DSPF[2] format) in flat or hierarchical forms, and supports "feed through" and "split" module ports in hierarchical DSPF.
- Includes automatic handling of signal coupling, wherever the DSPF parasitic data includes coupling capacitors between signal nets. This significantly increases the accuracy without sacrificing efficiency.
- Efficiently analyzes real and apparent loops in the design, including combinational and latch loops.
- Supports user-specified false paths and multi-cycle paths. Further, false paths can be qualified by general timing constraints, for increased robustness in false path removal.
- False paths can be validated using design functionality, under the *DesignCheck* option.

[2] Detailed Standard Parasitic Format

- Supports logic value specification and automatic propagation for additional false-path blocking.

- Supports general timing constraint specification through SDF[3] constraint annotation.

- Provides a range of reporting options, including critical paths, point-to-point delays, slacks and histograms.

- Outputs accurate SDF delays for use in other analysis and simulation tools.

- Includes an application-programming interface (API) for easily linking *TimingCheck* with other software systems such as logic synthesis tools, place-and-route tools, etc.

- Supports incremental timing analysis, when design changes are provided through the API.

- Supports the proprietary Timing Interface and Budgeting (TIB) models[4], for enhanced flexibility in modeling complex cells and functional units at various stages of a design cycle.

- Generates spice decks for selected critical paths, including all the interconnect parasitic data and loading. The critical paths are automatically sensitized statically, for easily running the simulations.

- Provides a powerful option to automatically optimize gate sizes and insert buffers to fix both setup and hold violations with accurate post-route parasitic information. Generates “ECO” information for use in incremental place-and-route changes.

2.2 Static Design Constraint Verification (DesignCheck)

Comprehensive logic and circuit verification of complex VLSI designs has become more critical than ever with the exploding design complexities enabled by current semiconductor technologies. As designs become larger and more complex, it also becomes increasingly difficult to achieve the same level of verification that was possible in earlier generation designs, unless the verification methodology keeps pace with the designs.

[3] Standard Delay Format
[4] In addition to standard timing models.

Existing verification solutions do not completely cover the verification space for complex ASIC and custom designs. There are numerous design constraints and requirements that are not verified exhaustively in many complex designs at present. These design constraints arise due to various design rules, requirements, restrictions, guidelines, specifications and assumptions that are common in most designs. These constraints must be met in order for the design to function correctly and reliably.

The constraints may be related to various verification domains such as functionality, timing, testability, reliability, circuit restrictions, or combinations and overlaps of these areas. In general, such design constraints often straddle multiple verification domains and usually require exhaustive verification over the entire design. Therefore, the verification tool must use static analysis techniques to provide exhaustive coverage. The tool should also be able to analyze multiple problem domains and interactions between them in a single framework.

There is clearly a need for a new design constraint verification (DCV) tool that can meet these requirements, complement existing tools and fit easily into most design methodologies. Since the design constraints may vary significantly between different designs, it is also necessary to allow users to "program" the tool, so that the verification space can be customized to suit each design.

SDA's *DesignCheck* option is a *static design constraint verifier capable of exhaustively analyzing large designs in acceptable run times*.

DesignCheck allows users to program various design rules, specifications, constraints, assumptions, etc., using a very simple language. It then exhaustively verifies the entire design for conformance to those requirements.

The constraints can be written to verify existence of certain properties, as well as non-existence of certain other properties. These properties are typically not completely verified in the course of normal functional and timing verification methodologies. The intent of constraint checking is to help enforce the fundamental rules of a particular design or design environment on a day-to-day basis as the design is created and modified. This prevents potentially serious design problems that may require costly analysis, debug and re-design late in the design cycle or after the chip is already in production.

DesignCheck is intended to be used together with existing verification tools, and is not meant as a replacement for any other tool. For example, it can work closely with timing verifiers and delay calculators, by means of importing Standard Delay Format (SDF) delays or signal arrival times from these tools. The uniqueness of this tool is that it can analyze and process topology, functionality, timing and electrical information in a single framework, so that potential interactions between these analysis areas can be considered in the verification.

The following are examples of typical rules and constraints that can be verified exhaustively using *DesignCheck*:

- Cell usage rules, including: functional relationships, timing phase relationships, timing skew, load limits and connectivity rules at inputs or outputs of a cell.
- Testability rules, including: scan chain connectivity, controllability/observability logic, test clock schemes, and mutual exclusion conditions to avoid logic conflicts.
- Clock gating/buffering and clock distribution rules.
- Clock phase rules between source and destination of a path, between inputs of a dynamic logic gate, etc.
- Detailed clock skew constraints within and between various regions in a global clock distribution network.
- General timing constraints, such as: skews at cell inputs, pulse-widths, delay of a path segment, skew between delays of two path segments, etc.
- Functional rules such as mutual exclusion of drivers at tristate buses, requirement that there should be a default driver on every tristate bus, etc.
- False timing path checks based on logic functionality.
- Signal coupling checks based on both timing and functional analyses.

3. STATIC TIMING ANALYSIS/VERIFICATION COMMANDS

This section describes the syntax and semantics for the user commands that control the timing analysis and verification. These commands must be provided to SDA in the form of a command file (specified by the “-cmd” command-line option).

3.1 Primary input (PI) or internal node arrival time and source clocking

(a) %pi_arr <pi_name> <min_rise_arr_time> <min_fall_arr_time>
<max_rise_arr_time> <max_fall_arr_time> <clk_wvfm_name>
<clk_edge>
[-trans <min_rise_trans_time> <min_fall_trans_time>
<max_rise_trans_time> <max_fall_trans_time>]

(b) %pi_arr <pi_name> <min_rise_arr_time> <min_fall_arr_time>
<max_rise_arr_time> <max_fall_arr_time> -async
[-trans <min_rise_trans_time> <min_fall_trans_time>
<max_rise_trans_time> <max_fall_trans_time>]

(c) %set_arr_time <internal_net_or_pin_name>
[<min_rise_arr_time> <min_fall_arr_time>
<max_rise_arr_time> <max_fall_arr_time>
<clk_wvfm_name> <clk_edge>]
[-trans <min_rise_trans_time> <min_fall_trans_time>
<max_rise_trans_time> <max_fall_trans_time>]

- *pi_name*: primary input pin/net name
  ⇒ could be a bit-select or part-select of a bus (using Verilog bus syntax)
  ⇒ '*' is a global wild-card to cover all PIs
- *arr_time*: arrival time in ps, accumulated externally from clock edge (L/T) time
- *clk_wvfm_name*: source clock waveform name defined by *wvfm* command
- *clk_edge*: launching edge of source clock - Leading (-L) or Trailing (-T)
- *async*: indicates an asynchronous input without a clock reference
- -trans: The optional transition times (waveform slopes) are assumed to be measurable at certain thresholds (such as 20% and 80%) consistent with the library characterization and the slope thresholds specifications in the SDA command file. (Default transition times are 0.)
- If an internal node is specified (through %set_arr_time), it is treated as a pseudo-primary-input and the arrival time is propagated forward from that node. Also, in this case, if arrival times are omitted (including clock waveform and edge), then only the transition times are applied to that node. The specified arrival times and/or transition times override any values computed internally for that node.

3.2 Primary output (PO) required time and destination clocking

(a) %po_req <po_name> <max _time> <min _time> <clk_wvfm_name>
<clk_edge>

(b) %po_req <po_name> <max_time> <min_time> -async

- *po_name*: top-level module pin (or net) name
  ⇒ could be a bit-select or part-select of a bus (using Verilog bus syntax)
  ⇒ '*' is a global wild-card to cover all POs
- *max_time*: absolute external required time, for worst-case condition, in ps

- *min_time*: absolute external required time, for best-case condition, in ps
- *clk_wvfm_name*: destination clock waveform name
- *clk_edge*: sampling edge of destination clock (-L or -T)
- *async*: indicates an asynchronous output without a clock reference

(Note that the required times are not “setup” or “hold” times, but absolute time points within the destination clock waveform.)

3.3 Primary output (PO) load

%po_load <po_name> <capacitive_load >

- po_name: top-level module pin (or net) name
  ⇒ could be a bit-select or part-select of a bus (using Verilog bus syntax)
  ⇒ '*' is a global wild-card to cover all POs
- *capacitive_load*: external capacitive load in ff

3.4 Clock waveform definition

%wvfm <wvfm_name> <pol> <Lmin> <Lmax> <Tmin> <Tmax> <period>

- *wvfm_name*: character string (must be unique for each wvfm cmd)
- *pol*: -R if Leading edge is rising; -F if Leading edge is falling
- *Lmin/Lmax*: leading edge of clock (min/max bounds) in ps
- *Tmin/Tmax*: trailing edge of clock (min/max bounds) in ps
- *period*: clock period in ps

3.5 Assigning waveform to clock net

(a) %pi_clk <pi_name> <wvfm_name>
[-R|-F min_rise min_fall max_rise max_fall]
[-trans <min_rise_trans_time> <min_fall_trans_time>
<max_rise_trans_time> <max_fall_trans_time>]

- Assign a clock waveform to a primary input.
- Clock property at a PI (primary input) overrides any pi_arr statement on the same PI.
- pi_name can be any primary input name.
- Optionally, the clock arrival times and polarity can be modified and specified immediately following the wvfm_name. The waveform will still be used for reference purposes, but actual arrival times and polarity will be based on the modified information specified here. This would be useful

in situations where the clock incurs some delay and possibly polarity change before arriving from an ideal source to the input of a design.

- -trans: The optional transition times (waveform slopes) are assumed to be measurable at certain thresholds (such as 20% and 80%) consistent with the library characterization and the slope thresholds specifications in the SDA command file. (Default transition times are 0.)

(b) %clk <net_name> <wvfm_name>
[-trans <min_rise_trans_time> <min_fall_trans_time>
<max_rise_trans_time> <max_fall_trans_time>]

- Assign a clock waveform to an internal net.
- *net_name* can be a any internal net name.
- -trans: The optional transition times (waveform slopes) are assumed to be measurable at certain thresholds (such as 20% and 80%) consistent with the library characterization and the slope thresholds specifications in the SDA command file. (Default transition times are 0.)

(c) %default_clk <wvfm_name>
[-trans <min_rise_trans_time> <min_fall_trans_time>

- Assign a default clock waveform to all “unclocked” flip-flop or latch clock pins, which do not derive clock information from “%pi_clk” or “%clk” commands specified at other nets.

3.6 Analysis options

(a) %set_condition [-min | -max | -min0|-max2|-maxp]

- Analyze either best-case (min) or worst-case (max) condition. For best-case analysis, the *min* delays will be used from the SDF data and the critical paths will be interpreted as the shortest paths between each pair of source and destination. For the worst-case analysis, the *max* delays will be used from the SDF data and the critical paths will be interpreted as the longest paths between each pair of source and destination.
- The -min0, -max2 and -maxp conditions allow signal coupling analysis to be done under one condition at a time, to support distributed NOD/NOSN analysis. -min0 specifies best-case coupling effect, -max2 specifies worst-case coupling effect, and -maxp (max’) specifies actual effect of coupling on the delay of victim nets. (For these options, signal-coupling analysis should always be enabled using %set_signal_coupling_analysis and related commands.)

(b) %latch_open_setup <setup_value>

- Global setup requirement for data arriving before opening clock edge of a latch, such that data does not pass through transparently. Default is zero.

(c) %false_path <A1> <A2> .... <An>

- Any path that touches the sequence of points A1 ... An (in the given order) is to be treated as a false path and removed from the analysis. The number of points can be just one or more than one. Each point Ai is a hierarchical net or pin name in the design.

(d) %multi_cycle_path [-min | -max] <A1> <A2> <addCycles>
%multi_cycle_path [-min | -max] -clk_domains <src_clk_wvfm_name>
<dest_clk_wvfm_name> <addCycles>

- A multi-cycle path is specified using a pair of hierarchical net or pin names, followed by the number of clock cycles (addCycles) by which the path setup or hold constraint is to be modified. The first point (A1) must refer to the output of a latch or flip-flop. The second point (A2) must refer to the input pin of a latch or a flip-flop or a primary output. 'addCycles' can be positive or negative, and uses the clock period of the destination clock waveform.

- The destination can be wild-carded using the "*" character, to apply the multi-cycle path property to all possible destinations from the specified source. Current restrictions: The source cannot be wild-carded, and the destination cannot be wild-carded partially.

- Multi-cycle paths can also be defined with a single point. All paths passing through this point will be treated as multi-cycle, and the cycle adjustment will be made to the arrival time at this point using the source clock waveform. The path report will include the notation " AM " to indicate an arrival-time adjustment for a single-point multi-cycle path. If the multi-cycle requires positive (negative) cycles to be added to shift the constraint edge to a later (earlier) time, then the arrival time will be shifted to an earlier (later) time for equivalency. Integral numbers of the *source* clock period will be used for this adjustment.

- Multi-cycle paths can also be specified in terms of clock domains (i.e., user-defined clock waveform names), rather than node names.

- By default, multi-cycle paths are defined for setup analysis only. Using the "-max" and "-min" options, they can be defined separately for setup and hold analyses. Note that a positive addCycles will increase the slack in a

setup analysis, while a negative addCycles will increase the slack in a hold analysis.

(e) %clk_gate_out <hierarchical cell instance name> <output pin name>

- This command sets a specified cell instance as a “clock gate”, where clock and data signals are being gated together. In such a case, the data signal will be prevented from propagating to the specified output pin of the cell instance. This is useful in conjunction with the following two commands. (Note: This command is no longer required, since clock-gating data signals are blocked automatically starting from version 1.6.)

(f) %setup_check <hierarchical_test_net_or_pin> -R|-F|-B
<hierarchical_ref_net_or_pin> -R|-F <setup_constraint>
%setup_check <hierarchical_test_net_or_pin> -R|-F|-B
<clk_wvfm_name> -L|-T <required_time>

(Notation:- L: lead ; T: trail ; R: rise; F: fall; B: both rise and fall.)

- This command performs a generic setup check between two nodes, or at any node with respect to a clock waveform and edge. When <required_time> is specified with respect to an ideal clock waveform and the waveform's leading/trailing edge, it is interpreted similar to a %po_req command. The <required_time> is interpreted as an absolute required time for MAX condition with respect to a destination clock waveform and clock edge.

(g) %hold_check <hierarchical_test_net_or_pin> -R|-F|-B
<hierarchical_ref_net_or_pin> -R|-F <setup_constraint>
%hold_check <hierarchical_test_net_or_pin> -R|-F|-B
<clk_wvfm_name> -L|-T <required_time>

(Notation- L: lead; T: trail; R: rise; F: fall; B: both rise and fall.)

- This command performs a generic hold check between two nodes, or at any node with respect to a clock waveform and edge. When <required_time> is specified with respect to an ideal clock waveform and the waveform's leading/trailing edge, it is interpreted similar to a %po_req command. The <required_time> is interpreted as an absolute required time for MIN condition with respect to a destination clock waveform and clock edge.

(h) %clk_gate_data_setup -cell_type [-cell_and|-cell_nand|-cell_or|-cell_nor]

testEdge refEdge <num>

- Specify globally a setup constraint for the data inputs of clock gates. The cell type is one of AND, NAND, OR, or NOR. ‘testEdge’ is -R or -F (indicating rising or falling edge of data). ‘refEdge’ is -R or -F (indicating rising or falling edge of clock). The last argument <num> is the setup constraint, indicating the amount of time (ps) that the data signal must be valid before the specified clock edge.

(i) %clk_gate_data_hold -cell_type [-cell_and|-cell_nand|-cell_or|-cell_nor]
testEdge refEdge <num>

- Specify globally a hold constraint for the data inputs of clock gates. The cell type is one of AND, NAND, OR, or NOR. ‘testEdge’ is -R or -F (indicating rising or falling edge of data). ‘refEdge’ is -R or -F (indicating rising or falling edge of clock). The last argument <num> is the hold constraint, indicating the amount of time (ps) that the data signal must be valid after the specified clock edge.

(j) %set_logic_value -H|-L <hierarchical net or pin name>

- This command sets a high (H) or low (L) logic value at the specified net or pin in the design. All timing paths that pass through this point will be blocked as false as a result of the fixed logic value.

(k) %prop_logic_values

- This command propagates any user-specified logic values through the combinational logic portion of the design. The logic value propagation is terminated when a storage element is encountered, or when the logic value at the output of a combinational element can not be determined unambiguously.

(l) %mux _out <hierarchical cell instance name> <output pin name>

- This command sets a specified cell instance as a “multiplexer” where the control signals (selectors) may be clocks. In these cases, both the controlling clock signals and the data signals are allowed propagate to the output of the cell, since critical paths may originate from any of these signals.

(m) %dynamic_gate_out <hierarchical cell instance name> <output pin name>

- This command sets a specified cell instance as a “dynamic gate. In these cases, both the clock signals and the data signals are allowed propagate

to the output of the cell, since critical paths may originate from any of these signals.

(n) %set_cell_prop [-dynamic|-mux] -cell <cellTypeName>

- Specify a particular cell *type* in the library as a dynamic gate or as a multiplexer. This is useful or needed when clock signals are used to control such gates. All instances of the specified cell type are covered by this specification.

(o) %set_path_search_limit <n>

- Limit the maximum number of paths searched per user-defined false path or user-defined reporting path. Default: 1000.

(p) %set_level_depth_limit <n>

- Limit the maximum number of levels to be traversed for levelizing the design. Default: 500.

(q) %set_looped_latch_depth_limit <n>

- Limit the number of cascaded transparent latches to be analyzed when the latches are part of loops. Default: 8.

(r) %set_infer_flip_flops

- Convert back-to-back latches (such as master-slave) into an equivalent flip-flop.

(s) %set_parallel_drvr_error_tol <n>

- Set the percentage error tolerance allowed in the alignment of input timing at parallel gates. (Specify n between 0 and 1, where 0 allows no error at all.) Default: 0.01 (1%).

(t) %set_parallel_drvr_error_tol_ps <n>

- Set the parallel driver error tolerance in time units (ps) instead of percentages.

(u) %set_zero_clock_delays

- Force zero delays through all clock buffers and clock nets.

(v) %set_parallel_drvr_equalize_delay [-cell_inst_list { cell_inst_name_1
cell_inst_name_2 ... }]

- Force parallel clock drivers to produce fully aligned clock arrival times at the output, provided that the original alignment tolerance of the parallel drivers was within the specified tolerance limit. The "-cell_inst_list" option can be used to group parallel drivers into multiple parallel groups: This way, all the parallel drivers driving a common net can be divided into multiple parallel groups based on the physical proximity of the drivers. If two groups of clock buffers are driving a major clock trunk from two ends, then these buffers can be separated into two groups. Within each group, the cell/interconnect delays are equalized and the delays are calculated with the parallel effect of all the drivers within that group taken into account.

(w) %set_select_fast_clock_paths

- Select the worst-case clock paths from multiple parallel distributed clock buffers.

(x) %set_no_min_max_interaction

- While using either the min condition libraries and/or RC scaling, this command allows the min and max analyses to be independent of (orthogonal to) each other, especially when setup/hold constraints are checked at various points in a design. Under this option, minimum clock path delays will be used to check hold-time constraints and maximum clock path delays will be used to check setup-time constraints, even though selecting the opposite-condition clock paths may result in a tighter setup or hold constraint. This may be necessary if process/temperature/voltage variations are not significant enough within the same die.

(y) %block_async_paths_thru_seq_elems

- Block any asynchronous timing path that passes through a sequential cell, such as a flip-flop or a latch. This includes paths through data input pins of the cell which do not have a setup or hold constraint to a clock input of the cell, and paths reaching clock input pins which do not originate from a clock source.

(z) %set_clock_skew -max|-min -clk_domains <src_clk_wvfm_name>
<dest_clk_wvfm_name> <skew>
%set_clock_skew –max|-min -global <skew>

- Clock skews can be defined between pairs of clock domains (i.e., user-defined clock waveform names), so that the final slack is reduced by the amount of the skew. Clock skews can be defined between two different clock domains, or within the same domain, by appropriately specifying both the source and destination clock waveforms. Clock skews can be defined separately for setup and hold analyses, by specifying the "-max" or the "-min" switch in the command.
- Clock skews can also be defined globally (i.e., between all possible pairs of clock domains, including within each domain), by using the "-global" version of the command. Also, the global skew command can be followed by commands to selectively set specific source-destination skews in particular cases.
- The reporting command "%rpt_clk_skew" will display these clock skew values at the end of each reported path.

3.7 Delay calculation options

(a) %set_internal_delay_cal [-elmore|-awe|-awe_mesh|-elmore_mesh]
[-ctot|-ceff|-ceffp]

- This command forces SDA to perform internal delay calculation at all stages where there is no SDF delay back-annotation for both cells and interconnect. The cell delay calculations are based on either linear or non-linear (table) models provided in the library, using the capacitance of the interconnect network as the output load. If DSPF parasitic data is not available for the output net of the cell, then the total pin capacitance from all the receivers on that net is used as the output load.[5] The detailed delay calculation options are:

  -elmore : Elmore method for RC tree delay calculation
  -awe : AWE method for RC tree delay calculation
  -awe_mesh : AWE method for RC tree or mesh delay calculation (the more expensive mesh algorithm is used selectively whenever the RC network contains loops)
  -elmore_mesh: 1-pole AWE method (Elmore-like) for RC tree or mesh delay calculation, using the total capacitance (ctot) for gate output loading

  -ctot : total interconnect capacitance applied to gate output (default)
  -ceff : effective capacitance applied to gate output
  -ceffp : average of ceff and ctot applied to gate output

[5] Also known as fanout capacitance.

Usage notes:

Most accurate and efficient combination of options : "-awe_mesh -ceff".
More conservative and efficient combination of options : "-awe_mesh -ceffp".
Do not use "%set_fast_drvg_point_slope" whenever -ceff or –ceff.

(b) %set_temperature_factor <temperature_value>

- This command sets an operating temperature value, so that the library based delay calculation can be scaled accordingly. The temperature value is an absolute value of the temperature, in units consistent with the library.

(c) %set_voltage_factor <voltage_value>

- This command sets an operating voltage value, so that the library based delay calculation can be scaled accordingly. The voltage value is an absolute value of the voltage, in units consistent with the library.

(d) %set_process_factor <process_value>

- This command sets a process corner value, so that the library based delay calculation can be scaled accordingly. The process value is an absolute value of the process corner, in units consistent with the library.

(e) %set_rise_slope_thresholds <startVal> <endVal>

- This command sets the starting and ending thresholds (as fractions) for measurement of waveform slopes in the rising direction. The threshold values are floating point numbers between 0 and 1. The default values are 0.2 and 0.5 for startVal and endVal respectively. Note that these values must be set according to the way that the waveform slopes are measured (for both input and output signals of a cell) during characterization of the library cells.

(f) %set_fall_slope_thresholds <startVal> <endVal>

- This command sets the starting and ending thresholds (as fractions) for measurement of waveform slopes in the falling direction. The threshold values are floating point numbers between 0 and 1. The default values are 0.8 and 0.5 for startVal and endVal respectively. Note that these values must be set according to the way that the waveform slopes are

measured (for both input and output signals of a cell) during characterization of the library cells.

(g) %set_inc_anal_err_tol <tolVal>

- This command sets the error tolerance for any incremental timing analysis iterations. The tolerance value (tolVal) is a floating point number between 0 (0%) and 1 (100%), with a default value of 0. It represents the errors in delays and waveform slopes that can be tolerated during any incremental analysis iteration. A higher value of tolVal will reduce the runtime for incremental analyses, and will produce correspondingly less accurate results.

(h) %set_fast_drvg_point_slope [-min|-max]

- This command allows the user to optionally force the driving point (cell output pin) waveform slope to be *fast* (a ramp of 10 ps) regardless of the actual slope calculated from the delay model, input slope and output capacitive load. This option is useful when the user requires less conservative results for interconnect delay values. This option also provides a simple mechanism to experiment with the effects of the driving point slope modeling. By default, this option is disabled. The user can select whether to apply this option to both min and max conditions, or to only one of them by selecting one.

(i) %set_rc_mesh_nodes_limit <number>

- User programmable limit on number of nodes in an RC interconnect circuit at any net, for the most accurate AWE mesh solution for interconnect delays. Default is 5000. When the limit is exceeded, warning messages are issued, and AWE mesh solutions are not used (AWE tree solutions are used instead) for specific nets that exceed this limit. The reason for this limit is that very large interconnect circuits (such as with 20000 nodes) require a large amount of virtual memory for processing.

(j) %set_hi_accuracy_rc_nodes_limit <number>

- User programmable limit on number of nodes in an RC interconnect circuit at any net, above which the most accurate interconnect delay solution is slightly compromised for efficiency. Default is 5000. The purpose of this limit is to allow the user to trade off a some accuracy for faster execution time. When this limit is exceeded at any net, accurate multi-pole AWE solutions will be replaced with 1-pole (Elmore-like) faster solutions.

(k) %set_global_lib_scale_factor <lib_scale_factor_name>

- Select one of many "scaling_factors" factors defined in the library, for purposes of scaling the library values prior to delay calculation. If any cell does not have a cell-specific "scaling_factors", then this global scaling factor will be applied to that cell for computing the effect of process/voltage/temperature variations on delay and transition time.

3.8 Path reporting options

(a) %rpt_mode -crit

- *crit*: Report critical paths (rise/fall) in global sorted order from all end points

(b) %rpt_mode -full <D1> <D2> .... <Dn>

- *full*: Report concisely all paths that end at destination points D1 .... Dn. The destination points must be primary outputs or synchronous inputs of storage elements, and specified as hierarchical instance names or pins.

(c) %rpt_mode -slack <N1> .... <Nn>

- *slack*: Report worst slack on hierarchical nets N1 ... Nn.

(d) %rpt_mode -worst_path

- *worst_path*: Report a single worst path regardless of its slack or any other criteria. The *crit* and *worst_case* modes are mutually exclusive, and the last one specified in the command file will override any previous specifications.

(e) %rpt_mode -hist -slack <N>

- *hist slack*: Generate a histogram of critical paths grouped by slack. N is the slack increment (in ps) to be used for path groups in the histogram. The maximum slack and the maximum number of paths must be defined separately, and these criteria will be used in selecting and grouping the paths for the histogram.

- The range of the histogram will be as follows:

start: smallest slack computed in the analysis

end: maximum slack defined by the *rpt_max_slack* command

(f) %rpt_mode -hist -freq <N>

- *hist freq*: Generate a histogram of critical paths that violate setup or hold constraints, grouped by frequency. The histogram will use the initial clock frequency as the maximum frequency, and reduce the frequency by increments of N (in MHz) until there are no violations. The histogram will be approximate in cases where there are transparent latches or multiple frequency clocks.

- The range of the histogram will be as follows:

  start: frequency used in the timing analysis
  end: estimated lower frequency where there is no violation

(g) %rpt_max_slack <n>

- Report only those critical paths that have slack less than n.

(h) %rpt_num_paths <n>

- Report the top n paths that meet the slack criterion.

(i) %rpt_long_names

- Reporting format to accommodate long hierarchical names.

(j) %rpt_path <A1> <A2> .... <An>

- Any path that touches the sequence of points A1 ... An (in the given order) is reported as a “user-defined path”. Each point Ai is a hierarchical net or pin name in the design. The number of points can be just one or more than one. If just one point is specified, then the critical paths ending at that point will be reported, and paths in this category will be sorted by increasing slack values. If multiple points are specified, then all possible critical path segments covering the sequence of points will be reported, and paths in this category will be sorted by decreasing path delays. Further, the path sorting is done considering all of the user-defined paths globally. *No paths will be reported if more than one of the specified points belong to the same net in the design.*

(k) %rpt_clk_skew

- Report the clock skew between source and destination registers/latches for each reported path.

(l) %rpt_long_path_min_level <n>

- Report long paths only if they have at least n levels. Default: 0.

(m) %rpt_short_path_max_level <n>

- Report short paths only if they have no more than n levels. Default: infinite.

(o) %rpt_src_module_inst_list { modinst1, modinst2, ..., modinstn }

- Report only paths that originate from one of the specified hierarchical module instances.

(p) %rpt_dest_module_inst_list { modinst1, modinst2, ..., modinstn }

- Report only paths that end at one of the specified hierarchical module instances.

(q) %rpt_path_node_details

- Both critical paths and user-defined paths can be printed out with additional details at each node on the path. These details include separate cell and wire delays, and the effective capacitance and resistance seen from the driver within that stage. The purpose is to provide additional information that may be useful in understanding and optimizing timing paths.

(r) %rpt_signal_coupling_details [-noise_per_pin]

- When signal coupling analysis is enabled, this reporting command (when used in conjunction with %rpt_path_node_details) provides the following additional information on how the maximum delays of victim nets have been adversely impacted (degraded) by the simultaneous switching of coupled aggressor nets:
  - “Pre-coupling delays” at affected victim nets within critical path timing reports, indicating what the maximum cell and net delays would have been in the absence of coupling.
  - “Coupling Report” at the end of the timing report file, listing each victim net that has been affected by the simultaneous switching of coupled aggressor nets. Also included: total wire capacitance of the

victim net (scaled to reflect coupling effect), and raw/scaled coupling capacitance to each aggressor net.
    - "Pre-coupling" SDF delays (within the SDF output file) for cell instances and interconnects that have been affected by the simultaneous switching of coupled aggressor nets, indicating what the maximum delays would have been in the absence of coupling.
- It also provides detailed reporting of noise-on-static-net violations, including peak noise voltages on victim nets. When the "-noise_per_pin" option is included in this command and the "-accurate_mode" option is used with the "%set_noise_on_static_net_analysis" command, the reporting includes peak noise voltages at each of the receivers on a victim net.

3.9 Using Statistical Wire Load Models

Statistical wire load models can be read in from the library, when the DSPF data is unavailable or only covers portions of a design. Then, wire load estimation can be enabled by the following command:

Syntax: %set_wire_load_estimation_mode [-module <module_name>]
[-lumped_cap | -lumped_pi_rc]
[-wire_load_model <model_name> |
-wire_load_selection <selection_name> -hier_depth <num>]

This command can be used multiple times to specify the combination of wire load models to be used from the library. There are three forms of this command:

- The "-wire_load_model" option specifies a single wire load model (named "model_name"), which is to be applied globally to the entire design. Without the "-module" option, this should only be specified once.
- If "-wire_load_model" is used in conjunction with the "-module" option, then the wire load model only applies to all instances of the specified module type. The wire load model also applies to all child module instances below the specified module instances in the design hierarchy. This form of the command can be used multiple times to specify different wire load models for different modules. All other modules will use either a global wire load model or area-based wire load models, as specified by other forms of the command.
- The "-wire_load_selection" option is to be used without either of the other options mentioned above. It specifies a "wire_load_selection" structure in the library that SDA should use to automatically apply cell-area-based wire load models for different modules. The "-hier_depth" option (required) specifies the hierarchical depth to which this option should apply; below this depth, the wire load models of parent modules will be applied recursively to all child modules in the hierarchy. This form of the command should only be used once, and will override any global wire load model specification.

In each of the above cases, the wire load estimation can be in the form of lumped capacitance or a lumped "pi" RC circuit, using the "worst case tree" assumptions.

The wire load model can be used in timing verification, timing optimization, SDF generation and timing rule checks. The RC networks created internally for the wire load model can also be written out in DSPF format (output_filename.dspf), using the command "%dspf_write -max".

4. SIGNAL COUPLING (NOISE) ANALYSIS COMMANDS

SDA includes an optional signal coupling analysis capability, which can analyze two types of coupling and noise effects using coupling capacitance represented in the DSPF parasitic data:

1. Noise-on-delay (NOD) analysis of simultaneous switching of coupled nets, resulting in “degraded” delays for victim nets that have been adversely impacted by such coupling. The NOD effect can be taken into account within the timing analysis automatically.
2. Noise-on-static-net (NOSN) analysis of noise pulses induced on quiet (static, or non-switching) nets due to switching activity at coupled aggressor nets. The purpose is to find significant amounts of noise induced on nets after they have already switched and settled into a final logic state for the current clock cycle. The NOSN analysis uses the results of the timing analysis, including all NOD effects, and can be done automatically following the timing analysis.

4.1 Enabling Noise-On-Delay Coupling Analysis

The following command enables noise-on-delay (NOD) coupling analysis for accurate delay calculation and timing analysis, wherever the DSPF parasitic data includes coupling capacitance between signal nets:

Syntax: %set_signal_coupling_analysis

The following additional commands allow further control of the NOD analysis:

Syntax: %set_coupling_capacitance_multipliers <Kmin> <Knom> <Kmax>
Syntax: %set_coupling_window_margins <Tmin> <Tmax>
Syntax: %set_small_coupling_capacitance_parameters <capRatio> <Kmax>
Syntax: %use_fine_grained_kfactors
Syntax: %set_mutex_switching_nets [-nod | -nosn]
<victim_netname> <aggr_netname>

The parameters set by the above commands are defined as follows:

- Kmin, Knom and Kmax are the factors by which a coupling capacitance value is multiplied for minimum, typical and maximum delay cases, after taking into account other information such as timing windows. Default values are: 0, 1.0 and 2.0.
- Tmin defines the minimum time that must elapse between the victim net's latest switching transition and the aggressor net's earliest, in order for SDA to see "no simultaneous" switching from the point of view of the victim.
- Tmax defines the minimum time that must elapse between the latest switching transition of the aggressor net and the latest transition of the victim net, in order for SDA to see "no simultaneous" switching from the point of view of the victim. Default values are: 0 and 0.
- CapRatio is the ratio of <coupling capacitance from one aggressor net to a victim net> to <nominal total capacitance of the victim net>. It defines a threshold under which coupling between a specific aggressor and a specific victim may be ignored, and the coupling capacitors may be grounded and multiplied by the Kmax factor. This allows small amounts of coupling to be ignored (either optimistically or pessimistically), so the analysis can be more efficient and focused on significant cases of coupling.
- The "%use_fine_grained_kfactors" command instructs SDA to convert the discrete "Kfactor" values (defined by Kmin, Knom and Kmax) into fine-grained values based on the relative strengths (drive strength + wire resistance effect) of the aggressor and victim nets. A weaker victim will have a higher Kfactor than Kmax, while a stronger victim will have a lower Kfactor than Kmax. This makes the NOD analysis more accurate and realistic.
- The "%set_mutex_switching_nets" command can be applied separately to NOD or NOSN cases. In each case, hierarchical net names for a victim and an aggressor must be specified. The corresponding effect of NOD (increase in delay) or NOSN (noise induced on victim net by aggressor) is removed as a result of this command. For NOD cases, this means that the victim and aggressor switch in mutually exclusive clock cycles and are therefore not subject to any simultaneous-switching delay increases. For NOSN, it simply means that the aggressor cannot induce noise on the quiet victim (for example, if the given aggressor is mutually exclusive with other aggressors for the given victim). In general, this command is meant to act in a way similar to false-path removal in timing analysis, allowing users to remove false NOD effects or NOSN violations.

4.2 Enabling Noise-On-Static-Net Coupling Analysis

The following command enables noise-on-static-net (NOSN) analysis based on the results of timing analysis and the locations of coupling capacitors between various signal nets:

Syntax: %set_noise_on_static_net_analysis
-accurate_mode | -fast_mode [-enable_heuristics]

The following additional commands allow further control of the NOSN analysis:

Syntax: %set_supply_voltage <supply_voltage>
Syntax: %set_noise_margin [-all | -cell <cell_type>]
<high_noise_margin> <low_noise_margin>
Syntax: %set_driver_resistance -cell <cell_type>
-pin <output_pin_name> <pullup_resistance> <pulldown_resistance>
Syntax: %set_mutex_switching_nets [-nod | -nosn]
<victim_netname> <aggr_netname>

The parameters and options set by the above commands are defined as follows:

- The “-accurate_mode” option uses an extension of AWE to compute the noise induced on quiet victim nets by switching aggressors. Accurate aggressor slew-rates are used in conjunction with this coupling model to compute noise pulses and evaluate noise violations on victim nets.
- The “-fast_mode” option uses the “coupling-enhanced pi-RC” model to compute the noise induced on quiet victim nets by switching aggressors. This model extends the well-known “pi” RC model (two capacitors and one resistor) by splitting each capacitor into a grounded capacitor and a coupling capacitor. Accurate aggressor slew-rates are used in conjunction with this coupling model to compute noise pulses and evaluate noise violations on victim nets. (This option is an alternative to the “-accurate_mode” option. If both “-fast_mode” and “-accurate_mode” are enabled in the same command statement, then SDA will use the “-fast_mode” option first, followed by “-accurate_mode” wherever violations have been detected.)
- The “-enable_heuristics” option enables certain heuristic algorithms to improve the accuracy of the “fast_mode” NOSN analysis.
- “supply_voltage” is specified in volts, with a default value of 3.0.
- “high_noise_margin” is specified in volts, and represents the maximum tolerable amount of low_to_high noise that can be induced on a victim net at the input of a receiver. It can be specified globally using “-all”, or at inputs of specific cell types using “-cell”. If the noise margin is specified on a cell basis, then it must beset for all cell input in the library.
- “low_noise_margin” is specified in volts, and represents the maximum tolerable amount of high_to_low noise that can be induced on a victim net at the input of a receiver. It can be specified globally using “-all”, or at inputs of

specific cell types using “-cell” (“-all” is an easy way to globally set a noise margin).

- “pullup_resistance” is specified in ohms, and represents the output pull-up resistance of the driver cell on a victim net. If this is not specified for a cell type, then SDA will infer this indirectly from timing characteristics.
- “pulldown_resistance” is specified in ohms, and represents the output pull-down resistance of the driver cell on a victim net. If this is not specified for a cell type, then SDA will infer this indirectly from timing characteristics.
- The “%set_mutex_switching_nets” command can be applied separately to NOD or NOSN cases. In each case, hierarchical net names for a victim and an aggressor must be specified. The corresponding effect of NOD (increase in delay) or NOSN (noise induced on victim net by aggressor) is removed as a result of this command. For NOD cases, this means that the victim and aggressor switch in mutually exclusive clock cycles and are therefore not subject to any simultaneous-switching delay increases. For NOSN, it simply means that the aggressor cannot induce noise on the quiet victim (for example, if the given aggressor is mutually exclusive with other aggressors for the given victim). In general, this command is meant to act in a way similar to false-path removal in timing analysis, allowing users to remove false NOD effects or NOSN violations.

4.3 Some Guidelines for Coupling/Noise Analysis

Include the following commands (in addition to standard timing analysis and reporting commands) for NOD analysis with emphasis on verifying long paths and setup constraints:

```
%set_coupling_capacitance_multipliers 0.0 1.0 2.0
%set_coupling_window_margins 0 0
%set_small_coupling_capacitance_parameters 0.1 1.0
%set_signal_coupling_analysis
```

This ignores any net-to-net coupling that results in less than 10% of the total capacitance on the victim net, and focuses the analysis on the more significant cases of coupling. The corresponding capacitors at the victim net are grounded and treated as normal capacitors. Other parameters (multipliers, margins) are at default values. (Note: “%set_condition” should not be used when coupling analysis is enabled.)

For additional reporting on degradation of maximum delays due to NOD, within both timing reports and output SDF delays, and also for detailed NOSN violation reports, include the following command:

```
%rpt_signal_coupling_details -noise_per_pin
```


It is recommended that SDF delays be written out (using "%sdf_write") for more information on pre-coupling and post-coupling maximum delays at cell instances and nets.

If the DSPF does not include duplicated coupling capacitors (i.e., if coupling capacitance is indicated at only one of any two coupled nets), then include the following additional command:

%duplicate_spf_cc

Note that the above command is required for a fully functional NOSN analysis; therefore, it is recommended that the DSPF file not include duplicated coupling capacitors.

For NOSN analysis, include the following set of additional commands:
%set_noise_on_static_net_analysis -accurate_mode
(or: %set_noise_on_static_net_analysis -fast_mode -enable_heuristics)
(or: %set_noise_on_static_net_analysis
-accurate_mode -fast_mode -enable_heuristics)
%set_supply_voltage
%set_noise_margin

In addition, the following commands could be used for additional accuracy and control:
%set_driver_resistance
%set_rise_slope_thresholds
%set_fall_slope_thresholds

The following is a typical command file for a combined timing and NOD/NOSN noise analyses:

```
/* Define timing for clocks, primary inputs, primary outputs */
%wvfm w1 -R 0.0 0.0 1000.0 1000.0 2000.0
%pi_arr in2 0 0 0 0 w1 -L
%pi_arr in1 5000 5000 5000 5000 w1 -L
%pi_arr in3 5000 5000 5000 5000 w1 -L
%po_req * 800 200 w1 –T

/* Timing analysis controls: delay calculation and NOD coupling analysis parameters */
%set_internal_delay_cal -awe_mesh -ceff
%set_coupling_capacitance_multipliers 0.0 1.0 2.0
%set_coupling_window_margins 0 0
%set_small_coupling_capacitance_parameters 0.1 1.0
%set_signal_coupling_analysis

/* NOSN noise analysis parameters */
%set_noise_on_static_net_analysis -fast_mode -enable_heuristics
%set_supply_voltage 3.0 /* 3.0 is also the default */
%set_noise_margin -all 0.75 0.75 /* noise tolerance levels for low-to-high and high-to-low
noise voltage pulses */
```



```
%rpt_signal_coupling_details  -noise_per_pin /* reports aggressor details for both NOD and
NOSN */
%duplicate_spf_cc  /* necessary for fully functional noise analysis */
%set_driver_resistance -cell inv_4 -pin y 10000 10000
%set_rise_slope_thresholds  0.2 0.8
%set_fall_slope_thresholds 0.8 0.2

/* Timing reporting */
%rpt_mode -crit
%rpt_max_slack 5000
%rpt_num_paths 20
%rpt_path_node_details
%rpt_mode -hist -slack 100
%rpt_ckt_stats

/* SDF output */
%sdf_write

/* Miscellaneous */
%set_multiple_output_files
```

As another example, for hold-time analysis, if coupling analysis were to be replaced with a simple 50% reduction of all coupling capacitors, then the following commands would be needed:

```
%set_small_coupling_capacitance_parameters 1.1 0.5
%duplicate_spf_cc       /* If DSPF does not include duplicated coupling capacitors. */
```

5. SPICE DECK GENERATION COMMANDS

SDA provides two types of spice deck generation capabilities: extraction of logic cones from any given node; and path spice decks related to critical and user-defined paths in the timing report.

5.1 Logic cone spice deck generation

Spice decks for logic cones can be generated by specifying the source node (net or pin) of the logic cone. A spice subcircuit is generated for the forward cone (till storage elements or primary outputs) from the source. Along with the subcircuit, additional information for delay measurement points and name cross-referencing is provided. Each cone is written out to a separate file (output_filename.cone_*i*.spice, where *i* is a unique index number for each file).

(a) %set_logic_cone_spice_deck_gen <hierarchical_net_or_pin>
[-prop { split_at_level <level> output_pins_first }]

The "output_pins_first" property enables printing output pins first. Default is input pins first. Also, the input or output pin order is taken from the cell instances in the design netlist. (See %spice_subckt command below.)

The level number (an integer) specified with the "split_at_level" tells SDA to split each fanout at the net at that level (counting the source node as level 1) into a different spice subcircuit. Each such spice subcircuit will include the common path from the source node. Each property is an option and can be either included or omitted within the braces. If "split_at_level" is not specified, then the entire logic cone will be included in one subcircuit.

If any parallel gates exist within the logic cones, they are *all* included in the spice netlists.

An application of this feature would be to generate a spice deck for a clock tree (entire tree or portions of it), for use in accurate spice simulation of delays. Logic cone spice decks can be split into multiple subcircuits and files (to reduce size of circuit to be simulated).

5.2 Path spice deck generation

(a) %set_path_spice_deck_gen –crit_paths [-all | <num>]
(b) %set_path_spice_deck_gen –rpt_paths [-all | <num>]

Spice deck generation can be enabled for *critical paths* (max/setup paths and/or min/hold paths, depending on the use of %set_condition) by using (a); and for *user-defined paths* (both max and min paths) by using (b). User-defined paths are defined as usual by using %rpt_path.

All spice subcircuits are written out to a single file (output_filename.spice). All the max/setup subcircuits are listed first, followed by the min/hold subcircuits, the user-defined full-path subcircuits, and finally the user-defined path-segment subcircuits – these four categories are listed in the same order as in the timing report. Within each category, the subcircuit number matches the corresponding path number.

In each type of syntax above, the last field is either the *number* of top paths for which spice subcircuits are required, or “-all” implying that spice subcircuits are required for *all* reported timing paths. In all cases, the number of subcircuits in each category is limited by the actual number of paths reported within that category in the timing report.

All gates in these subcircuits are sensitized such that the one input of interest can switch and propagate the transitions through the path. Off-path capacitive and RC loading is included with each path for correct loading.

If any parallel gates exist along a path, they are *all* included in the spice netlist provided that non-identical gates are not placed in parallel. The user must still check the SDA log file to make sure that there is no timing misalignment at the inputs of a group of parallel gates. The parallel gates are all sensitized in the same way. The start of a path (primary input, including clocks; or internal node) must occur at least one logic stage prior to the inputs of any parallel gates – otherwise, the connectivity at the inputs of the parallel gates may be incorrect when RC data has been back-annotated to the input nets.

5.3 Spice subcircuit pin order definition

Library cell “pin order” can be defined to match the corresponding spice subcircuit definitions, so that the spice decks generated by SDA are compatible with the subcircuit definitions.

Syntax: %spice_subckt <cell_name> [<pin1_name> <pin2_name> …]

The pin order can be defined this way for any number of library cells. In the case of multi-bit (bussed) pins, each individual bit of the pin must be listed separately using the format “pin_name[index]”.

Any unused output pin in the cell instance will be connected to a dummy net named “UNUSED”. As before, any “side input” pin which could not be set to a logic value will be connected to a dummy net named “VXX”.

This command will impact the spice decks generated for paths and logic cones. If the pin order is undefined for any library cell, then the default is to use the pin order from the cell instantiation in the Verilog netlist, with inputs or outputs listed first as chosen by the user.

6. TIMING OPTIMIZATION COMMANDS

SDA provides timing optimization capabilities which are tightly integrated with the static timing analysis engine. Based on user commands, SDA can re-size gates (up or down) and insert buffers (including cascaded buffers for optimality). The timing optimization capabilities (also known as ECO capabilities) can be used to fix timing problems that are causing failures or poor performance under both the worst-case (max) and best-case (min) conditions. The user may optionally define the sizing/buffering parameters (if default values are not acceptable); the user

must select the type of optimizations required, by using the commands in section 6.2.

SDA iteratively attempts to fix timing problems, subject to the command specifications discussed below, and incrementally updates the timing information. The final timing report printed out reflects all the sizing/buffering changes made to the design. The modified design is written out as a Verilog netlist (output_filename.v) and as a set of changes to be used in incremental place-and-route (output_filename.eco.size and/or output_filename.eco.buffer).

6.1 Sizing and Buffering Parameters

The user-controllable sizing and buffering parameters are defined as follows, with the sizing and buffering parameters controllable independently:

(a) %set_sizing_parameters <maxcap_to_ceff_ratio> <load_to_intrinsic_del_ratio>
<load_to_interconnect_del_ratio> <mincap_to_ceff_ratio>

(b) %set_buffering_parameters <maxcap_to_ceff_ratio> <load_to_intrinsic_del_ratio>
<load_to_interconnect_del_ratio>
<pre_buffer_derate_factor>
<bufferchain_sizeup_factor>
<wire_resistance_threshold_for_repeater>

maxcap_to_ceff_ratio

Minimum limit for the ratio:
(maximum capacitance that a cell can drive) /
(effective load capacitance that the cell is actually driving).

If this ratio is less than maxcap_to_ceff_ratio, then SDA will attempt to size up the cell or insert a buffer. A larger value of maxcap_to_ceff_ratio will result in more aggressive optimizations.

This ratio can be defined separately for sizing and buffering optimizations, and is used with the “-max_cap” optimization constraint.

Default = 1.5.

total_to_intrinsic_del_ratio

Maximum limit for the ratio:
(total delay of a cell) / (intrinsic delay of a cell).

This ratio must exceed total_to_intrinsic_del_ratio in order for SDA to perform sizing or buffering optimzations to fix timing slack for paths passing through the cell. A smaller value of total_to_intrinsic_del_ratio will result in more aggressive optimizations; however, a value of less than 1.0 is not recommended.

This ratio can be defined separately for sizing and buffering optimizations, and is used with the "-min_slack" optimization constraint.

Default = 1.5.

total_to_interconnect_del_ratio

Maximum limit for the ratio:
(total delay of a cell) / (interconnect delay at the output of a cell).

This ratio must exceed total_to_interconnect_del_ratio in order for SDA to perform sizing or buffering optimizations to fix timing slack for paths passing through the cell. A smaller value of total_to_intrinsic_del_ratio will result in more aggressive optimizations; however, a value of less than 1.0 is not recommended.

This ratio can be defined separately for sizing and buffering optimizations, and is used with the "-min_slack" optimization constraint.

Default = 1.5.

mincap_to_ceff_ratio

Maximum limit for the ratio:
(maximum capacitance that a cell can drive) /
(effective load capacitance that the cell must drive).

(This ratio is no longer used.)

pre_buffer_derate_factor

When buffers are inserted, the first buffer inserted at the output of an existing cell instance is sized such that its input capacitance is no more than:
(maximum capacitance that a cell can drive) / (pre_buffer_derate_factor).

A smaller value of pre_buffer_derate_factor will typically result in fewer cascaded buffers at the output of any cell instance.

Default = 4.0.

bufferchain_sizeup_factor

This parameter defines the maximum ratio by which successive buffers in a cascaded buffer chain must be sized up. A larger value of bufferchain_sizeup_factor will typically result in fewer cascaded buffers at the output of any cell instance.

Default = 1.5.
(This factor is no longer used.)

wire_resistance_threshold_for_repeater

Minimum wire resistance value which triggers an "advisory" message in the log file when sizing or buffering failed to meet optimization criteria at any particular cell instance and its output net. The advisory message recommends that the net is a potential candidate for repeater insertion.

Default = 500 (ohms).
(This factor is no longer used.)

If one of the above commands is not used, then the default values will be assigned to the corresponding set of sizing and/or buffering parameters. If some (but not all) of the parameters in either of the above commands is to be set to a non-default value, then all of them must be set. The default value can be assigned to any parameter which is not to be modified.

6.2 Setup and Hold Optimizations

(a) Syntax for setup-time optimization:

%optimize_gate_sizes [-max_cap] [-min_cap] [-min_slack <num>]
[-max_trans <num>] [-max_trans_opt <num>]
[-enable_buffer_insertion] [-no_netlist_output]
[-improve_ff_setup]

- The “-max_cap” switch enables sizing changes for improving signal rise/fall times. The optimization is done based on the “maxcap_to_ceff_ratio” parameter discussed earlier.
- The “-max_trans” switch also enables sizing changes for improving signal rise/fall times, but uses the specified maximum transition time as a constraint. When specified, this switch overrides the “-max_cap” switch. Note that the transition time value must be consistent with the thresholds used for measuring rise/fall times in the library and specified using %set_rise_slope_thresholds and %set_fall_slope_thresholds. These

thresholds are also used in reporting waveform “slopes” in timing reports. Since only one transition time value can be specified as constraint, the rise and fall constraints must be the same.

- The “-max_trans_opt” switch enables both sizing and buffering changes for improving signal rise/fall times, using the specified maximum transition time as a constraint. When specified, this switch overrides both the “-max_cap” and the “max_trans” switches. Unlike “-max_trans”, this option uses delay optimization techniques to generate more optimal solutions, and can be used with or without “-min_slack”. Note that the transition time value must be consistent with the thresholds used for measuring rise/fall times in the library and specified using %set_rise_slope_thresholds and %set_fall_slope_thresholds. These thresholds are also used in reporting waveform “slopes” in timing reports. Since only one transition time value can be specified as constraint, the rise and fall constraints must be the same.
- The “-min_slack” option allows specification of a minimum slack value, below which a path is considered to be in violation and therefore a candidate for optimization. The default is 0.
- The “-enable_buffer_insertion” switch enables selective insertion of buffers in instances where size changes alone cannot meet the minimum slack constraint.
- The “-no_netlist_output” switch disables the generation of an output Verilog netlist.
- The “-improve_ff_setup” enables additional resizing of flip-flops to reduce setup constraints, if the cell library is designed for such optimization.

(b) Syntax for hold-time optimization:

```
%optimize_gate_sizes  -fix_hold_time [ -min_hold_slack <min-hold-slack> ]
                                     [ -min_setup_slack <min-setup-slack>]
                                     [-no_netlist_output]
```

- The option "-min_setup_slack" allows the user to provide a minimum setup slack, which must be preserved at any storage element input while fixing hold times. This setup slack can be positive or negative. The intent is to avoid adding too many buffers to fix hold-time violations in cases where setup time may be inadvertently impacted due to small setup slacks. The default is to add as many buffers as needed to fix hold-time violations.
- The “min_hold_slack” switch defines a minimum hold-time slack, below which a path is considered to be in violation and therefore a candidate for optimization.
- *Note: Hold-time ECO should be run separately from setup ECO (i.e., following all setup ECO changes), but with both min and max libraries and DSPF with RC scaling.*

(c) Size changes:

- Cell size changes use interchangeable sets of cells based on their "footprint" property. (Functional similarity is assumed when the footprint is the same.)

(d) Syntax for selecting a prototype cell type for buffer insertion:

%use_buffer_cell_type <cellType>

- If cellType is valid (i.e., if it is a non-inverting cell, with a single input and a single output, and having a valid “footprint” property), then the family of cells with the same "footprint" property will be considered as candidates for use in adding buffers for both minslack/maxcap constraints and for hold-time constraints.

(e) Syntax for excluding certain cell instances from optimization:

%dont_optimize <cell-instance-name>

- Exclude the specified hierarchical cell instance name from any sizing or buffering optimization. The instance name can contain wildcards (“*”), but all the hierarchy separators (“/”) should be present. It is recommended that a relatively small number of cell instances be specified in this manner.

(f) Syntax for excluding certain cell instances from optimization:

%dont_use <library-cell-name>

- Do not use the specified library cell for any sizing or buffering optimization. The cell name can contain wildcards (“*”).

6.3 Some Guidelines for Optimization

1. Commands for setup optimization through gate sizing only:

```
%set_condition -max
%optimize_gate_sizes  -max_cap  -min_slack <value>   -no_netlist_output
```

2. Commands for setup optimization through gate sizing and buffer insertion:

```
 %set_condition -max
%use_buffer_cell_type  <representative-buffer-cell>
                        /* specify any buffer cell from the buffer cell family */
%optimize_gate_sizes  -max_cap  -min_slack <value>
                        -enable_buffer_insertion   -no_netlist_output
```


3. The “-max_trans” switch may be used instead of “-max_cap” for setup optimization. In general, for fixing the rise/fall times, it is better to rely directly on slack optimization than on through either of these switches. In most cases, slack optimization will also fix any rise/fall violations.

4. Commands for hold optimization through buffer insertion:

```
/* do not use "%set_condition" -- let SDA default to both min and max analysis */

%use_buffer_cell_type <representative-buffer-cell>
                        /* specify any buffer cell from the buffer cell family */
%optimize_gate_sizes  -fix_hold_time  -min_hold_slack <value>
                        -min_setup_slack <value>  -no_netlist_output
```

5. For hold optimization, the following are recommended for more accuracy in both the setup and the hold timing numbers:
   a. Use two sets of libraries (slow and fast) concurrently in the same SDA run. (Use both -lib and -libmin switches in the command line).
   b. Scale down the R/C values for a more conservative hold analysis, while leaving the R/C intact for setup.
   (Example: "%set_rc_scale_factors –min -res 0.5 -cap 0.5 -no_clk_rc_scaling").

6. Generally, retain the “-no_netlist_output” switch in all optimization commands, unless an output Verilog netlist is required.

7. The buffer cell type must have the following properties: non-inverting, single-input, and single-output. It must also have the "footprint" property specified in the library.

8. Do not set the sizing and/or buffering parameters, unless there is a specific need to change the default settings.

9. When using the results of any ECO run, the "*.eco.size" file must be applied first, followed by the "*.eco.buffer" file. The second file assumes that the changes in the first one have already been applied to the design, so this sequence is important. If only one of these files is generated, then, of course, that single file can be applied directly.

10. Always verify that the basic timing analysis is correct in terms of clocks, primary outputs, primary inputs, false paths, etc., by looking at the "*.pre_opt" output file.

7. STATIC CONSTRAINT VERIFICATION COMMANDS

DesignCheck can be used to exhaustively verify static design constraints and rules in a wide range of areas in typical VLSI designs. These constraints and rules are categorized into various generic *classes of rules* as described below. Under each class, specific constraints and rules pertaining to a particular design can be written (or "programmed") by the user using the *rules language* described in this section. The rules can be written using cell types, cell instances, net names, net types or generic latch and flip-flop types, so that the rules can be as general or as specific as desired. This results in a customized verification space to suit each design.

DesignCheck will normally produce informational messages to indicate that each constraint or rule in the command file is being checked.[6] Whenever there is a violation, *DesignCheck* will also output specific information to describe the violation and will identify the paths, cells and/or nets that caused the particular violation.

This section describes the syntax and semantics of the language that users can utilize to write rules and constraints specific to their designs. This user-specified information is provided to *DesignCheck* as part of the *command* file. (In the following documentation, any optional arguments or parameters are enclosed in square brackets. In cases where one out of several arguments must be used, a slash is used to separate the available choices.)

7.1 Cell Usage Verification

Syntax: ***%cell_check [-id ruleName]***
cell_specification -prop { properties }
[pin_direction] [pin_specification]

The cell_check command can be used to verify symbolic functional constraints, symbolic timing constraints, and topological/structural constraints[7], by specifying constraints that relate to the immediate environment of a cell instance or all instances of a cell type. These constraints ensure that cells are instantiated and used in a manner consistent with their design, particularly from the perspective of how the cell inputs and outputs are connected and used.

The "-id" option allows the user to specify a character string to identify a particular rule or constraint. This identifier is echoed back into the output reports, along with the rest of the rule specification.

- *cell_specification*

[6] The terms "constraint", "rule" and "command" may be used interchangeably in this document.
[7] Using the terminology defined in the document "Static Design Constraint Verification: A New Solution".

⇒ ***-all*** : all cell instances in the design
⇒ ***-cell <cell_name>*** : all instances of specified cell type
⇒ ***-cell -inst <cell_inst_name>*** : hierarchical cell instance
⇒ ***-except <name>*** : exclude a specific cell type or cell instance from the constraint check (depending on whether "*-cell*" or "*-cell -inst*" is specified); typically useful when wild-carded names are used, and some specific names must be excluded.

- *properties*

  Each property in the following list must be used separately, and can not be combined with any other property in the same statement.

  ⇒ ***no_flt_input*** : check cell inputs for any floating inputs (default).
  ⇒ ***no_flt***: check for unconnected cell pins (inputs or outputs)
  ⇒ ***valid/invalid -net_type <net type>*** : specified cell pins should (should not) be connected to a net of the given type; generic net types may be one of the following:
    ◊ ***primary_input*** : all primary input nets
    ◊ ***primary_output*** : all primary output nets
    ◊ ***tristate*** : all nets driven by tristatable drivers
    ◊ ***multi_driver*** : all nets driven by multiple drivers
    ◊ ***logic_high*** : all nets with a fixed value of logic high
    ◊ ***logic_low*** : all nets with a fixed value of logic low
  ⇒ ***same_phase*** : specified cell pins are of the same clock phase.
  ⇒ ***opp_phase*** : specified cell pins are of opposite phases.
  ⇒ ***diff_phase*** : specified cell pins are driven by different clock waveforms.
  ⇒ ***equ*** : specified cell pins are driven by logically equivalent signals.
  ⇒ ***inv*** : specified cell pins are driven by inverted signals.
  ⇒ **mut_excl** : specified cell pins are driven by mutually exclusive (but not necessarily inverted) signals.
  ⇒ ***max_fanout <N>*** : the maximum number of receivers connected to a cell output is within the specified limit.
  ⇒ ***max_capacitance <N>*** [ff | pf | nf | f]: the total capacitance driven by a cell output is within the maximum specified limit (N is a floating-point number representing the maximum allowable capacitance).
  ⇒ ***max_transition <N>*** : the rise or fall time at the cell output is within the maximum specified limit (N represents a 0 to 100% rise/fall time in pico-seconds).

- *pin_direction*

  The optional pin_direction is specified as follows:

⇒ ***-input*** : used with *valid/invalid net_type*, or *no_flt*, to indicate a constraint on an input pin.
⇒ ***-output*** : used with *valid/invalid net_type*, or with *no_flt*, *max_fanout*, *max_capacitance or max_transition,* to indicate a constraint on an output pin.

- *pin_specification*

  If pin names are not specified, then all pins at the given cell will be checked. However, for rules that require a pair of input pins, both pin names must be explicitly specified. In addition, some properties can be associated with a *list* of pins names (which must all be inputs of the specified cell type or instance). The pin name can include wildcards only with the "*-pin*" option.

  ⇒ ***-pin <pin_name>*** : single-pin specification is used optionally with *no_flt, no_flt_input*, *net_type*, *max_fanout max_capacitance* and *max_transition* properties.
  ⇒ ***-pin1 <first_pin_name> -pin2 <second_pin_name>*** : pin-pair specification is used with *same_phase*, *opp_phase*, *inv, mut_excl* and *equ* properties where the relationship between two input pins is verified (in these cases, the pin names are required).
  ⇒ **-pinlist { pinName1 pinName2 ... }** : pin-list specification is used with *equ* and *mut_excl* properties where the relationship between multiple input pins is verified.
  ⇒ **-except –pin <pin_name>** : exclude the specified pin name from a "*-pin*" specification or when no pin name has been explicitly specified; may be used multiple times to specify a list of pin names to exclude.

7.2 Module Level Verification

Syntax: ***%module_check [-id ruleName]***
-module <module_name>
-prop { properties } -cell_list { list of cell names }

The module_check command, in its current form, is used to verify that a given module only instantiates cells from a given list of cell names, or that it does not use any cells from the given list. Further, the check can be done for all combinational cells or all sequential cells.

- *properties*

  One of the following properties must be specified:

⇒ ***valid combl*** : check only combinational cells and verify that they belong to the specified cell list.
⇒ ***valid seq*** : check only sequential cells and verify that they belong to the cell list.
⇒ ***valid*** : check all cells and verify that they belong to the cell list.
⇒ ***invalid*** : check all cells and verify that they do not belong to the cell list.

- *cell_list*

This is simply a list of cell names (types), separated by white space.

Syntax: ***%module_check [-id ruleName]***
-src -module –inst <moduleInstName>
-prop { find_path }
-dest_module_inst_list { inst1, inst2, …}

This command prints out all signals driven from the source module instance, to any one of the destination module instances. Each instance is a hierarchyical name string.

7.3 Functional Verification

Syntax: ***%func_check [-id ruleName] -prop { properties_list } net_specification***

The func_check command can be used to verify symbolic functional constraints, by specifying functional or signal-correlation rules at certain nets or pairs of nets. These rules verify certain required functional properties (such as mutual exclusion of tristate drivers, existence of a default driver on a tristate net, etc.). The rules can also be written to verify functional correlation relationships between pairs of signal nets, for applications such as signal coupling analysis.

- *properties*:

The valid or invalid property is used in conjunction with one of the other properties. The *mut_excl_drivers, similar_drivers* and *default_driver* properties must be used only with *valid*. Other correlation properties may be used with *valid* or *invalid*.

⇒ ***valid*** : verify that the other properties are valid.
⇒ ***invalid*** : verify that the other properties are invalid.
⇒ ***mut_excl_drivers*** : all tristate drivers on the specified net are enabled by mutually exclusive signals.
⇒ ***default_driver*** : there is a default driver on the specified tristate net under all conditions; the default driver may be one of the tristate drivers, or it

may be a weak pullup, holding circuit, etc., as defined using a *set_cell_prop* command.

⇒ ***similar_drivers*** : all drivers on the specified net are of the same cell type.

⇒ ***scpc*** : "same cycle positive correlation", meaning that the specified pair of nets may both switch in the same cycle but will not switch in opposite directions.

⇒ ***scnc*** : "same cycle negative correlation", meaning that the specified pair of nets may both switch in the same cycle and possibly switch in opposite directions (i.e., when one is switching from low to high, the other may switch from high to low in the same clock cycle)

⇒ ***dcc*** : "different cycle correlation", meaning that the specified pair of nets will never switch together in the same clock cycle.

⇒ ***scahc*** : "same cycle active-high correlation", meaning that the first net is in a logic high state when the second net (which could be a tristate net) is being actively driven.

⇒ ***scalc*** : "same cycle active-low correlation", meaning that the first net is in a logic low state when the second net (which could be a tristate net) is being actively driven.

⇒ ***thru_pipeline_stages exact <num>*** : specifies the sequential depth of the enable signals for tristate drivers (i.e., the number of latches or flip-flops between the combinational logic and the enable input of tristate driver cells) – to be used only in conjunction with the *mut_excl_drivers* and *default_driver* properties.

- *net_specification*

⇒ ***-net <net-name>*** : hierarchical net name, used with *mut_excl_drivers* or *default_drivers*.

⇒ ***-net_type <net-type>*** : generic net type, used with *mut_excl_drivers* or *default_drivers*, specified as *tristate* or *multi_driver.*

⇒ ***-net <net-name1> <net-name2>*** : a pair of hierarchical net names, used with *scnc, scpc, dcc, scahc*, and *scalc*.

⇒ ***-net <net-name1> -net <net-name2>*** : same as above.

⇒ **-pinlist { <pin-name1> <pin-name2> … }** : a list of hierarchical pin names and/or net names, to be used specifically with "valid mut_excl_drivers" property only (wildcards are permitted within the name strings).

7.4 Timing Correlation Verification

Syntax: ***%coupling_check [-id ruleName]***
<first_net_name> <second_net_name>

The coupling_check command can be used to verify value-based timing constraints. It can determine whether there is any overlap between the switching

time windows of the two nets, using the earliest and latest switching times (also known as arrival times) of signals at each net. The main application of this check is in signal coupling analysis. Either SDF delay data or arrival-time data must be back-annotated in order to use this check.

Restriction: The *coupling_check* command can not be used with *path_check/func_check/cell_check* in the same command file at present.

7.5 General Timing Constraint Verification

The SDF timing constraints supported by *DesignCheck* are compatible with SDF Version 3.0[8], and are part of the value-based timing constraint verification available in *DesignCheck*. The following constraint types, known as "Timing Check Entries" and "Timing Environment Entries" in SDF terminology, are currently supported within an SDF file:

- ⇒ ***SETUP, HOLD, SETUPHOLD*** : setup and hold constraints between inputs of a cell type or instance.
- ⇒ ***RECOVERY, REMOVAL, RECREM*** : recovery and removal constraints between an asynchronous input of a cell (such as reset) and a reference input such as a clock.
- ⇒ ***SKEW*** : maximum delay constraint between two inputs of a cell type or instance.
- ⇒ ***WIDTH*** : minimum pulse-width constraint at a clock input of a cell type or instance.
- ⇒ ***NOCHANGE*** : no-change constraint between a control input (such as a write-enable input to a memory) and a non-control input (such as address or data input to the memory) of a cell type or instance.
- ⇒ ***PATHCONSTRAINT*** : constraint (limit) on the maximum length (delay) of a given path segment.
- ⇒ ***SUM*** : constraint on the sum of the delays over two or more global paths.
- ⇒ ***DIFF*** *:* constraint on the difference between the delays over two global paths.
- ⇒ ***LESS****:* constraint that the delay of one global path is less than the delay of another path.
- ⇒ ***SKEWCONSTRAINT*** : constraint on the spread of delays from a common driver to all its driven inputs.

Restriction: The SDF timing checks must be specified in an SDF file, rather than the command file. This can be done using an additional SDF file, so that delay back-annotation and constraint specification can be separated. SDF timing

[8] Please refer to "Standard Delay Format Specification", Version 3.0, May 1995, published by Open Verilog International.

checks can not be performed along with *path_check/func_check/cell_check* at the same time.

7.6 Clock Skew Verification

Syntax: %clk_group [-local] <group_name> < net_or_pin_name> [-rpt_paths]
Syntax: %clk_skew_check [-id ruleName]
<group1_name> <group2_name> <max_skew>

The clock skew constraints supported by *DesignCheck* are part of the value-based timing constraint verification. A clock "group" is defined by specifying a hierarchical net or pin name within the clock distribution network in a design. All storage elements clocked downstream from this point will become part of this group. Each such group can be given a unique group name. If the "-local" option is used, then only the storage elements that are connected directly to the specified net, or the storage element clock pins that are specified directly, will be included in the clock group. The -rpt_paths option can be used to print out the longest and shortest paths to the worst-case clock destination points within a clock group.

Following this, clock skew constraints (maximum skew) can be specified within a group or between two groups. For intra-group skew constraints, the two group names in the constraint statement above will be the same.

7.7 Logic Value Based Verification

Syntax: %set_logic_value -H|-L < net_or_pin_name>
Syntax: %prop_logic_values
Syntax: %logic_value_check -H|-L < net_or_pin_name>

The *set_logic_value, prop_logic_values* and *logic_value_check* commands together verify value-based functional constraints. *set_logic_value* allows a logic value, high (H) or low (L), to be set at any hierarchical net or pin in the design. Once values have been set at various points in the design, the *prop_logic_values* command propagates the logic values through combinational logic as far as deterministically allowed by the functionality of the logic. Note that all logic values must be set initially prior to propagation, and the propagation can only be done once.

Then, *logic_value_check* command determines if the logic value at the specified hierarchical net or pin is high (H) or low (L), as specified in the rule. This type of rule is useful in verifying the state of various points (nets/pins) in the design, given the state of a few known points.

7.8 Functional False Path Verification

Syntax: %false_path_check [false_path_points]

The *false_path_check* command, which is part of the symbolic functional constraint verification capability of *DesignCheck*, can be used to verify whether a given path is functionally invalid. The path can be a full path or a partial path segment. The current version uses the static sensitization criterion.

- *false_path_points*

  This is a complete, ordered sequence of nodes in the path segment, where each node is a hierarchical cell pin or a primary input/output net. The verification will ignore any cell output pins, and use the sequence of input pins to evaluate the validity of the path.

7.9 Structural Path Validity Verification

Syntax: %path_check [-id ruleName] -prop { properties_list } path_points

The path_check command verifies symbolic timing constraints and topological/structural constraints. It allows users to specify rules for large groups of paths in a very compact way. Cell names or generic latch/flip-flop elements can be used to construct rules, so that specific cell instance or net names would not be typically required. The path_points describe the possible sources and destinations of the paths covered by each rule. In addition, the path_points allow the user to exclude certain sources or destinations from being checked. The properties are optional, and can be used to qualify the rule checking in many ways. If one or more properties are specified, then only the paths that satisfy those properties, along with the source and destination criteria, will be candidates for the rule check.

- *properties*

  ⇒ ***invalid*** : invalid path.
  ⇒ ***valid*** : valid path (default property).
  ⇒ ***excl_dest [num <N>]*** : exclusive for the destination – valid path where the destination can not have any source except the one specified. The optional “num” specification ensures that exactly N *source* path_points (each defined by a separate “-src”) exist for all the destinations taken together.
  ⇒ ***excl_src [num <N>]*** : exclusive for the source – valid path where the source can not have any destination except the one specified. The

optional "num" specification ensures that exactly N *destination* path_points (each defined by a separate "-dest") exist for all the sources taken together.

⇒ ***inv*** : inverting intermediate logic.
⇒ ***ninv*** : non-inverting intermediate logic.
⇒ ***combl*** : any combinational intermediate logic.
⇒ ***ncombl*** : no combinational intermediate logic.
⇒ ***combl_loops*** : combinational loop logic.
⇒ ***same_phase*** : source and destination are clocked by the same clock phase -- can be further qualified as valid or invalid path.
⇒ ***opp_phase*** : source and destination are clocked by opposite clock phases -- can be further qualified as valid or invalid path.
⇒ ***diff_phase*** : source and destination are clocked by different clock waveforms -- can be further qualified as valid or invalid path.
⇒ ***thru_pipeline_stages max / exact / min <N>*** : path may pass through sequential elements (flip-flops and/or latches) -- the number of sequential elements on the path can be specified exactly, or as a maximum/minimum bound; if the minimum bound is specified, then the maximum must also be specified in a separate *thru_pipeline_stages* property in the same path_check statement.

- *path_points*

⇒ ***-src <source_point>*** : starting points of a group of paths
⇒ ***-except -src <source_except_point>*** : these points are excluded from the set of valid starting points
⇒ ***-dest <destination_point>*** : ending points of a group of paths
⇒ ***-except -dest <destination_except_point>*** : these points are excluded from the set of valid ending points
⇒ ***-inter <intermediate_point>*** : intermediate points of a group of paths
⇒ ***-except -inter <intermediate_except_point>*** : these points are excluded from the set of valid intermediate points
⇒ ***-except <stop_point>*** : abort path searching when the stop_point is reached
⇒ ***-seq_xover_via <xover_point>*** : while crossing a sequential element (latch or flip-flop), the specified "crossover points" must be used

- Specifying *source_point, source_except_point, destination_point, destination_except_point, xover_point* and *stop_point.*

All of these "path points" can be specified in one of the following forms:

⇒ ***-cell <cell_name> [-pin <pin_name>] [-num <N>]*** : cell type, with an optional pin name, and an optional number of occurrences (all input or output pins will be used if pin name is not specified)

⇒ ***-cell_type <cell_type>*** : generic cell type -- at present, the only generic cell type supported is ***cell_inv***, which represents any inverter cell.

⇒ **-inst <cell_inst_name> [-pin <pin_name>]** : hierarchical cell instance name, with an optional pin name

⇒ ***-latch [-pin <pin_name>] [-num <N>]*** : generic "latch" cell type, specifying all latches in the design, with an optional generic pin name and an optional number of occurrences (number of occurrences can not be used with *xover_point*)

⇒ ***-ff [-pin <pin_name>] [-num <N>]*** : generic "flip-flop" cell type, specifying all flip-flops in the design, with an optional generic pin name and an optional number of occurrences (number of occurrences can not be used with *xover_point*)

⇒ ***-net_type <net_type>*** : generic net of the following types

- ◊ ***primary_input*** : all primary input nets
- ◊ ***primary_output*** : all primary output nets
- ◊ ***tristate*** : all nets driven by tristatable drivers
- ◊ ***multi_driver*** : all nets driven by multiple drivers
- ◊ ***logic_high*** : all nets with a fixed value of logic high
- ◊ ***logic_low*** : all nets with a fixed value of logic low

⇒ ***-net <net_name>*** : hierarchical net name

⇒ ***-net (-cell <cell_name> -pin <pin_name>)*** : specifying a group of nets indirectly, as the nets connected to a specific pin of all instances of the cell type

- Semantics, restrictions and recommendations on the usage of path_check

  The following semantic conventions, restrictions and recommendations must be observed in order to use the path_check command effectively:

  ⇒ Whenever the *same_phase* or *opp_phase* property is specified, it should preferably be accompanied by the *invalid* property. This can be typically used for detecting all same phase paths which may be invalid, for example. The *valid* property can also be used, although it may consume excessive runtime when large cones of logic must be searched. Also, these phase properties can only be used with generic storage elements (*-ff* or *-latch*) as the source and destination path points, although the *-except* path points can be specified using cell types, instances, or nets. The phase properties can not be combined with other structural properties, except for *valid* and *invalid* properties.

⇒ The *-except <stop_point>* specification is useful in filtering out false violations by stopping the path search at the specified path point. It effectively prunes the search space for the verification.

⇒ The *-except -dest* (or *–src* or *-inter*) specification removes the specified destination, source or intermediate path point from the overall rule specification, so that false violations can be filtered out.

⇒ The *–inter <intermediate_point>* specification can be used to specify intermediate cells, cell instances or nets which qualify the path. Only combinational cells or cell instances should be specified as intermediate points – sequential elements should be handled through the *-seq_xover_via* specification. *Valid* and *invalid* properties are checked only on paths that pass through specified intermediate points. *Excl_src* or *excl_dest* properties enforce the exclusive existence of such specified intermediate points, and will report violations if other types of intermediate points are found on any path under consideration.

⇒ The *-seq_xover_via* specification is required to cross any sequential element during path search, when also using the *thru_pipeline_stages* property. The specific pins of the sequential elements must be specified for selective sequential crossing through preferred pins.

⇒ The *thru_pipeline_stages* property is only applicable to the verification of exclusive paths.

⇒ The *-num* specification is only applicable to the verification of valid or exclusive *combinational* paths, and can be used to constrain the number of instances of valid source or destination elements (cell types or generic storage element types).

Besides the conventions and restrictions highlighted above, there are only a fixed number of meaningful or useful combinations for the various path properties. Table 1 (below) summarizes the combinations that are allowed or strongly recommended in order to obtain meaningful results. Each row in the table represents one of the legal combinations, with the X's identifying the properties that can be used together.

Table 1 : Legal combinations of *path_check* properties

invalid	*valid*	*excl_ dest*	*excl_ src*	*inv*	*ninv*	*combl*	*ncombl*	*combl _loops*	*same_ phase*	*opp_ phase*	*thru_ pipeline_ stages min*	*thru_ pipeline_ stages max*	*thru_ pipeline_ stages exact*
X									X				
X										X			
	X								X				
	X									X			
X								X					
		X		X							X	X	
		X		X								X	
		X		X									X
		X			X						X	X	
		X			X							X	
		X			X								X
			X	X							X	X	
			X	X								X	
			X	X									X
			X		X						X	X	
			X		X							X	
			X		X								X
X				X		X							
X				X			X						
X					X	X							
X					X		X						
	X			X		X							
	X			X			X						
	X				X	X							
	X				X		X						
		X		X		X							
		X		X			X						
		X			X	X							
		X			X		X						
			X	X		X							
			X	X			X						
			X		X	X							
			X		X		X						

7.10 Slack Verification

Syntax: ***%slack_check -low_slack <num> -high_slack <num>***

The slack_check command can be used to analyze the slack distribution at all nets in the design. If some (one or more) of the pins at a net have a slack less than or equal to the low_slack value (all time values are in ps), and if some pins at the same net have a slack equal to or greater than the high_slack value, then that net is reported along with the lowest slack values of all the pins at that net. The result may be useful in such applications as timing optimizations that are able to take advantage of the existence of both timing-critical and non-critical pins at certain nets.

7.11 Specifying Timing Environment and Timing Values

Timing information, such as clock waveform definitions, arrival times of primary inputs and clock phases from which primary inputs originated, is required for some types of constraints. For example, any phase checking rule, at the path level or the cell level, requires clock waveform and primary input clock phase information. The timing correlation rules, for coupling analysis, require the arrival times of primary inputs to be specified. General timing constraint verification also requires clock and primary input timing to be specified. Please refer to Section 3 for details on how to specify timing information and enable timing analysis.

7.12 Additional Options

There are a few additional options and global settings that can be selectively included in the command file, to further customize the verification.

- ***%rpt_mode -elaborate :*** This command specifies that, in certain functional and cell checks where Boolean functional analysis is performed, detailed functional information must be printed out to the log file on each violation. The default is to report only the violations without the detailed information.

- ***%set_cell_prop [-weak_holder | -pullup | -pulldown] -cell <cell-name>*** This command specifies that a particular cell type is a weak holder or a pullup or a pulldown circuit. This information is used to determine if there is a default driver on a tristate net (under the func_check command).

- ***%rpt_max_slack <N> :*** This command specifies the maximum slack threshold to use when reporting violations of SDF-based general timing constraints. Any slack that is less than N will be considered a violation. N is an integer representing timing slack in ps.

- ***%library_check [-id ruleName]*** : This command prints out all the library cells that were read in, along with detailed information on how sequential cells (latches, flip-flops, and memories) were interpreted. This is useful in verifying the correctness of newly created libraries.

8. LOGIC EXTRACTION AND PARTITIONING COMMANDS

SDA offers several mechanisms for partitioning the logic gates and nets in a design, and extracting the resulting piece of logic. This includes generating logic cones, interface logic and timing-driven clusters. By default, all of these output netlists (in Verilog or Spice formats) are written to the main output file. When multiple output files are chosen, the outputs go to the filenames specified below.

8.1 Basic Logic Extraction

(a) %intfc_logic_gen [-inputs | -outputs | -inouts] [-extract_clk_fanout]
[-verilog|-spice] [-except <io_net_name>]

- Extract all combinational logic from I/Os up to internal registers; optionally extract logic only from inputs or outputs or inouts; optionally extract clock trees and all fanouts of clock nets; and write out in Verilog or Spice formats (default: Spice). Output netlist is flat and is written to output_filename.intf.v (or .spice). In addition, if DSPF data was read in, then a reduced DSPF file corresponding to the output netlist can be written out using the "%dspf_write –max" command.
- The "-except" option can be used multiple times to exclude specific primary input/output nets, and associated combinational logic and registers, from the output netlist.

(b) %conn_logic_gen [-inputs | -inputs -pin pin_name | -outputs] cell_inst_name
[-extract_clk_fanout] [-verilog|-spice]

- Extract combinational logic cone from inputs and/or outputs of a cell instance; optionally extract clock trees and all fanouts of clock nets; and write out in Verilog or Spice formats (default: Spice). Output netlist is flat and is written to output_filename.conn.v (or .spice). In addition, if DSPF data was read in, then a reduced DSPF file corresponding to the output netlist can be written out using the "%dspf_write –max" command.

(c) %full_logic_gen [-spice|-verilog] [-long_names]

- Write out the *entire* design in Verilog or Spice formats, for any debug purposes. Output netlist is flat and is written to output_filename.full.v (or .spice). If the spice format is chosen, then an additional option (-long_names) specifies that full hierarchical names must be used in the spice netlist – the default is to use shorter names followed by a name cross-reference section at the end of the spice netlist. In addition, if DSPF data was read in, then the DSPF corresponding to the output netlist can be written out using the "%dspf_write –max" command.

8.2 Logic Partitioning and Clustering

(a) %enable_logic_partitioning [-min_size <n>] [-max_size <n>] [-min_area <n>]
[-max_area <n>] [-inc_ta] [-clean_up_top_level]
[-optimal <n>] [-preserve_names]
[-max_io_count <n>] [-follow_hierarchy]
[-single_clock] [-max_logic_depth <n>]
[-nom_logic_depth <n>]
[-incl_clk_cells] [-no_timing]

- Enable timing-driven logic partitioning and clustering. A given design netlist is repartitioned based on timing and size/area constraints. The result is a new Verilog netlist consisting of the newly partitioned modules.

- The command options are as follows:
  - min_slack: minimum timing slack (in ps) which qualifies a net for serving as a cluster boundary
  - min_size: minimum cluster size in number of cell instances
  - max_size: maximum cluster size in number of cell instances
  - min_area: minimum cluster area (based on cell and estimated wire areas from library)
  - max_area: maximum cluster area
  - inc_ta: perform incremental timing analysis after cluster creation (not recommended if "-optimal 5" is being used)
  - optimal <n>: various levels of optimization for cluster sizes
  - optimal 5: optimize cluster sizes, cluster I/O count and total runtime, and utilize the original design hierarchy to guide the partitioning process
  - optimal 6: dynamically selects the best partitioning algorithm, based on the characteristics of the input netlist, to obtain the best possible results, especially to minimize cluster I/O counts and top-level nets (recommended option)

- cleanup_top_level: include any cell instances left over at the top level into clusters (except for clock buffers)
- max_io_count: maximum number of inputs and outputs per cluster (not recommended if “-optimal 4 or –optimal 5” is being used)
- preserve_names: use original hierarchical instance and net names within each cluster and for top level nets – the name strings reflect the full original hierarchy, with a Verilog escape character (“\”) at the beginning – no name cross-reference is generated under this option
- follow_hierarchy: utilize the original hierarchical structure of the design, at the second level of the design hierarchy, to improve the quality of the partitioning result (recommended option for structured designs with good hierarchies)
- single_clock: attempt to create clusters such that each cluster is clocked by a single clock input
- max_logic_depth <n>: attempt to create clusters such that the combinational logic depth in each cluster is limited by the specified number “n”
- nom_logic_depth <n>: specify a “nominal” combinational logic depth of “n”, to guide the partitioning process in breaking up unstructured combinational logic (recommended option for unstructured designs that do not have good hierarchies)
- incl_clk_cells: include any clock buffers and clock gates within the clusters (default is to leave all cell instances, whose outputs contain clock timing events, outside of the clusters at the top level)
- no_timing: perform partitioning without using timing analysis

- The output files are as follows:
  - Verilog netlist with new partitions: output_filename.pc.v
  - Timing constraint file: output_filename.pc.constr
  - Information file (includes name cross-reference): output_filename.pc.info

(b) %set_non_partitionable_module_inst <hier_module_inst_name>

- Indicate a module instance which should not be flattened or partitioned. Such a module instance must be found at the second level of the original design hierarchy.

(c) %set_non_partitionable_cell_inst <hier_cell_inst_name>

- Indicate a cell instance which should not be included within any partition. Such a cell instance can be specified at any level of the original design hierarchy.

(d) %set_max_net_delay <hier_net_name> <max_delay>

- Specify a delay constraint (in ps) at any net in the design, to guide the clustering process.

(e) %set_cluster_boundary <hier_net_name>

- Force a net to serve as a cluster boundary, regardless of timing slack.

(f) %set_net_delay_to_length_factor <factor>

- Set a conversion factor to convert net delay (in ns) to wire length (in mm). If this factor is set, the output constraint file will include net length constraints in addition to net delay constraints. “factor” is a floating-point number.

(g) %set_cluster_naming <prefix> <start_index>

- Define the naming scheme for the new partitions (cluster modules) generated by SDA. “prefix” is a string, and start_index is an integer. Cluster modules are named in the form *prefix_index,* with increasing index values starting from start_index.

(h) %set_cluster_hier_divider <divider-character>

- Define the hierarchy divider character to be used in all of the outputs generated during logic partitioning (timing report, Verilog netlist, constraints, and information files). This command should only be used in conjunction with %set_new_escaped_name_rules.

(i) %keep_intact_module_inst <hier_module_inst_name>

- Indicate a module instance that should not be broken up during partitioning. All cell instances within these module instances will remain within the same cluster; however, each cluster may contain cells from multiple module instances. (Note: If this command specifies a module instance at some level of hierarchy, then it should not specify another module instance at a lower level within the same hierarchy. Also, this command should not be used along with the “-max_logic_depth” option.)

8.3 Some Guidelines for Partitioning and Clustering

8.3.1 Restrictions on valid inputs

- All non-partitionable blocks (NPBs) must be present at the second level of the original design hierarchy (i.e., must be instantiated at the top level of the design). If this is not possible, then such NPBs should be modeled as a blackbox in the library.

- All non-blackbox NPBs instantiated at the top level must have unique *module names*; otherwise, the output netlist will contain duplicate module definitions for these NPBs.

- If no routing-based parasitics (in DSPF format) are available for an NPB, then SDA may be used beforehand to generate estimated DSPF, based on a wire load model. This is done in a separate SDA run by selecting a wire load model for the NPB and then including the "%dspf_write –max" command in the SDA command file. The resulting output DSPF file can be used as one of the input DSPF files in the partitioning/clustering run of SDA.

- All partitionable logic must preferably consist of control and random logic, using typical standard-cell libraries. It is reasonable to also include datapath logic that has been generated through standard logic synthesis. Structured datapath is not appropriate for partitioning – these modules must be either black-boxed or specified as NPBs.

- It is recommended that the input Verilog netlist not include any clock buffers. Clock buffers will not be included in any of the clusters and will appear at the top level of the output netlist. It will be more appropriate to design the clock distribution after the clusters have been generated. Alternately, if the design already includes clock buffers and clock gates prior to partitioning, then the "-incl_clk_cells" option can be used to put all such cell instances into various clusters; then any critical clock buffers can be forced to be outside of the clusters by using the %set_non_partitionable_cell_inst command.

8.3.2 Getting the best results

- The design should have sufficient positive slack prior to partitioning. If the design does not meet timing prior to partitioning and physical design, then it is important to fix the timing at this stage (using an appropriate wire-load model) before proceeding further.

- It is recommended that, for each design, multiple partitioning runs be done using different parameters and constraints, so that the best result can be selected prior to embarking on physical design. This is important because different constraints may have different final impacts on the results, which cannot be predicted in advance. It is highly recommended that such a multiple-run approach be incorporated into the design flow.

- The min_slack constraint is used for initially breaking up the design and bounding the logic into several clusters. A large value of min_slack implies that a large slack threshold must be met before a net can serve as a cluster boundary (the net must have sufficient slack, greater than the threshold, to qualify as a cluster boundary net). A large value of min_slack also generally increases the value of the maximum net (pin-to-pin) delay constraints, making it easier to design interconnects between clusters. *The value of min_slack should be selected to be the smallest possible (positive) value that will still allow interconnect routing between clusters – this will give SDA more flexibility in partitioning the design and usually yield the best partitioning result.*

- The max_size and max_area constraints are mainly used for deciding to prevent two clusters from merging if the resulting size would be too large. They should be set to the largest acceptable values. SDA will attempt to satisfy these constraints only after meeting the min_slack constraint.

- The min_size and min_area constraints are mainly used for deciding when to merge small clusters with other clusters to form larger clusters. They should be set to the smallest acceptable values. SDA will attempt to satisfy these constraints only after meeting the min_slack constraint.

- **Always use the "–optimal 6" option to enable the latest improvements and enhancements to SDA's partitioning algorithms**. Command statement template:

```
%enable_logic_partitioning -optimal 6  -min_slack X  -max_size Y  -min_size Z
                           -preserve_names
```

(-max_size and –min_size can be replaced with –max_area and –min_area respectively)

- The following options (under optimal 5) should always be used separately and not combined in any one partitioning run: -follow_hierarchy, -nom_logic_depth, -max_logic_depth, -single_clock.

- For designs with a good structured second-level hierarchy and sufficient positive slack, the "-follow_hierarchy" option is highly recommended for best results under optimal 5. SDA will then attempt to preserve the second-level hierarchy as much as possible, subject to slack and size constraints. In such cases, the following complete command statement can be used as a template:

```
%enable_logic_partitioning -optimal 5  -min_slack X  -max_size Y  -min_size Z
                           -preserve_names  -follow_hierarchy
```


This option will attempt (if other constraints permit) to keep together all *connected* logic within each second level module instance. However, certain *independent* (i.e., unconnected) logic slices or groups from the module instance may eventually fall into different clusters depending on other (slack, size, etc.) constraints. Therefore, any given cluster may finally contain cells from different second level module instances.

- For designs that are flat and/or unstructured (in terms of hierarchy), or in cases where the slack is insufficient, the "-nom_logic_depth" option is recommended under optimal 5. It can initially break up the design better and allow SDA more flexibility in constructing the final clusters. In such cases, the following command statement can be used as a template:

```
%enable_logic_partitioning -optimal 5 -min_slack X  -max_size Y  -min_size Z
                           -preserve_names  -nom_logic_depth D
```

  It is generally useful to try multiple partitioning runs with variations in the depth number "D" (such as between 3 and 6, for example).

- The "-max_logic_depth" option can be used under optimal 5, as follows, if there is a need to strictly limit the number of levels of combinational logic on any path within any cluster:

```
%enable_logic_partitioning -optimal 5  -min_slack X  -max_size Y  -min_size Z
                           -preserve_names  -max_logic_depth D
```

  Even if the maximum depth cannot be limited in all instances, SDA will still attempt to limit the depth as much as possible everywhere. However, this is generally an artificial constraint which may not produce good quality results in many cases, and as such, it is not recommended unless it is imperative to limit the logic depth.

- The "-single_clock" option can be used under optimal 5, as follows, if there is a need to ensure that each cluster uses no more than one clock signal:

```
%enable_logic_partitioning -optimal 5  -min_slack X  -max_size Y  -min_size Z
                           -preserve_names  -single_clock
```

  Even if this constraint cannot be met in all instances, SDA will still attempt to enforce it as much as possible in all clusters. However, this is generally an artificial constraint which may not produce good quality results in many cases, and as such, it is not recommended unless it is very essential. Also, this option works best when each flip-flop or latch in the design is clocked by a single clock (i.e., no flip-flops or latches with multiple clocks).

- Do not insert clock buffers (and even scan-related logic, if possible) prior to cluster generation. Otherwise, the clock buffers will be placed at the top level of the output netlist (outside of all the clusters).

- There is no longer a need to use the "-inc_ta" option to perform incremental timing analysis. SDA outputs the correct timing report without this option. However, SDA no longer internally modifies the wire-load model to account for potentially longer inter-cluster interconnects, due to the fact that it now outputs the more accurate pin-to-pin delay constraints for these interconnects instead of the previous net delay constraints (since it is not possible to properly manipulate the wire-load parasitics at a particular net to account for widely varying pin-to-pin delays on the same net).

- Timing optimizations (ECO), including both setup-time and hold-time should be done separately, using SDA or other tools, in a separate run after the cluster generation. It is recommended that this be done after the physical design is complete, so that gate resizing and buffer insertion can be done in the presence of real parasitics.

- DSPF data can be selectively used for NPBs, while a wire-load model is used for the rest of the design. Such a combination will work *only* when the "%set_non_partitionable_module_inst" statement is included in one of the SDA command files (even in the case of a simple timing analysis without any partitioning). Also, note that the DSPF annotation applies only to the internal nets of NPBs; the wire-load model will be applied to all input/output nets of such blocks, since these nets connect to other logic outside these blocks.

- Primary input arrival times ("%pi_arr") and primary output required times ("%po_req") should be set conservatively, with sufficient timing margin built in, so that any external wiring outside the design can be accomplished without violating overall timing. SDA's partitioning will be based on these conservative arrival and required times specified by the user.

8.3.3 Design methodology

All of the partitionable logic must be synthesized using a single wire-load model for the partition size. This wire-load model is the same as the one used by SDA for the partitionable logic. SDA reads in the synthesized gate-level netlist of the design and performs partitioning and constraint generation. As pointed out earlier, it is very important that multiple partitioning solutions be explored at this stage. This implies running SDA multiple times with different constraints, so that

the best overall solution may be selected prior to physical design. Since it is not possible to predict the feasibility or quality of any particular solution in advance, a good design flow should explore the design space in this manner in order to obtain the highest quality results.

After a feasible partitioning solution has been generated by SDA and selected/approved by the user, the physical design process starts. The physical design includes top-level floorplanning, block/cluster placement, pin assignment, inter-block/cluster routing, and detailed intra-cluster place-and-route. There may be iterations between place-and-route and top-level pin assignment/routing.

Partitioning may be performed at the full-chip level (with NPBs or black-boxes for datapath and other non-partitionable parts of the design), or within a large functional unit/module. At either level, a decision has to be made as to how the partitioned design (chip or module) will be implemented physically. There are basically two different methods of approaching the physical design problem:

- The “hard” approach: Develop a floorplan of the partitioned logic, perform block placement for the clusters, pin assignment, etc., and then detailed place-and-route within each block (cluster). This approach may be particularly suitable for designs that have a good second-level hierarchical structure and have been partitioned using the “-follow_hierarchy” option.

- The “soft” approach (especially useful if partitioning has been done within a functional module): Directly perform detailed place-and-route within the functional module, with the clustering information used to group each cluster into a soft region. This could still accomplish most of the goals of partitioning, but without the extra effort required in the “hard” approach. In particular, in cases where the design (chip or functional module) does not have a good hierarchical structure, the resulting clusters are likely to have a higher number of inputs/outputs (for a fixed number of cells within a cluster), and therefore the “soft” approach may provide an easier path for implementation.

9. TIMING MODEL AND CONSTRAINT GENERATION COMMANDS

SDA includes the capability to generate an input/output timing model for a given design that is being timing-analyzed. The timing model is written to the main output file by default. When multiple output files are chosen, the model is written to the file *output_filename.lib*. SDA also has the ability to output accurate I/O timing constraints for black-boxed module instances at the top level of a design.

9.1 Commands

(a) %io_timing_model_gen [-max | -min]

- Generate an input/output pin-to-pin timing model of a design. The “-max” option generates a model based on the “max” condition delays and includes setup constraints for inputs. The “-min” option uses the “min” condition delays and includes hold constraints for inputs. This command must be accompanied by the appropriate timing analysis commands. Please refer to the application note for more details.

(b) %io_constr_gen

- Generate input/output timing constraints (consisting of arrival times, required times and clocks at the I/O ports, in SDA’s timing constraint language as defined in this user guide) for each module instance in a design. This is intended to be used only when the entire design has been previously partitioned and blackbox library models have been generated for each partition using %io_timing_model_gen or other means. When the design is analyzed again using only these models for all the partitioned modules at the top level, then accurate timing constraints can be generated for each such module instance and written out to a separate file for each instance. The constraints for each module instance is written out to a separate .cmd file based on the instance name of the module, and can be used for a separate, detailed timing analysis of the module.

9.2 Some Guidelines for Model Generation

1. SDA's timing model generation feature is meant to be used on normal data path blocks (modules) to create black-box timing models. These data path blocks are also referred to as "non-partitionable blocks" or NPBs in other documentation.

2. The model generation is not designed to be used on memory or other custom blocks (for which users should provide separate timing models). The model generation is not recommended for use on random/control logic, which should be typically included in the partitionable logic.

3. The SDA command file must be set up as in a normal timing analysis, with the addition of the '%io_timing_model_gen' command. The '-max' or '-min' option in this command should be consistent with the '%set_condition' command in order to produce meaningful results. SDA should be run on the block for which a timing model is desired. SDA will produce the model output in the 'output_filename.lib' file, in addition to other output files. It might be helpful to first run a successful SDA timing analysis; once this

has been done with the proper commands, then the '%io_timing_model_gen' command can be added to the command file.

4. The clock signals must be defined using '%pi_clk' at the primary inputs. Clocks should not be defined at internal nodes using '%clk'.

5. Clock waveforms must be defined using '%wvfm' such that each '%pi_clk' command has its own unique waveform name and the '%pi_arr' statements have one or more waveform names that are distinct from those used in '%pi_clk' commands. Such distinct clock waveform names are required even if some of the actual waveforms are identical in other respects.

6. The '%po_load' command can be used to set external capacitive loads at primary outputs, to represent wire capacitances or fanout capacitances that are outside of the block-to-be-modeled. In addition, the internal primary output nets can be annotated with DSPF parasitics.

7. In order to preserve timing accuracy, SDA internally flattens all buses into individual bits. Therefore, if the block-to-be-modeled has bussed I/O pins at its top level, then the SDA-generated timing model for this block must be instantiated at the upper level of the overall design using the following rule: *If the block-to-be-modeled has an input/output bus named 'A[2:0]' at its top level, then the SDA-generated timing model of this block must be instantiated at the upper level by using explicit pin names, which are not buses, such as 'A_2_', 'A_1_' and 'A_0_'.*

10. MISCELLANEOUS COMMANDS

(a) %sdf_write

- Output pin-to-pin cell and interconnect delays in SDF format, reflecting the effects of input SDF data that may have been read in and any internal delay calculation that may have been performed during the timing analysis. (Output file: output_filename.sdf.)

(b) %dspf_write [-max | -max2 | min] [-fix_index] [-flat]

- Output a modified DSPF file, with all coupling capacitors grounded and scaled correctly. The "-max" option scales the coupling capacitors accurately, provided that timing analysis and coupling (signal integrity) analysis have been enabled – otherwise, the capacitors are not scaled,

but simply grounded. The “min” option scales the coupling capacitors for the best-case, and “max2” scales the coupling capacitors for the worst-case – both without using the results of timing/coupling analysis. (Output file: output_filename.dspf.)

- The “-fix_index” option forces the index portion (following the delimiter “:”) of each subnet name to be unique within a DSPF NET section. This may be useful when the DSPF generated by SDA is read into other tools.
- The “-flat” option writes out the DSPF assuming that the Verilog netlist has been flattened. It should only be used along with the “%set_new_escaped_name_rules” command.

(c) %set_multiple_output_files

- This command forces SDF, DSPF, ECOs and spice decks to be written out to special files, with filename extensions .sdf, .dspf, .eco.size/buffer/v and .spice, respectively. The preceding part of the file names is the same as the name specified with the "-out" option on the command line. All other outputs (timing reports, static constraint verification reports, logic partitioning, etc.) are written out to the main output file.

(d) %duplicate_spf_cc

- If the DSPF file contains coupling capacitors that are not represented at both nets that each coupling capacitor is connected to, then the following command can automatically duplicate coupling capacitors internally. This is required for correct timing and other analyses involving signal coupling.

(e) %rpt_ckt_stats

- Report a summary of statistics on the design analyzed. This summary is printed at the end of the output file.

(f) %set_struc_depth_limit <limit>

- SDA’s default logic depth for “structural” path or cone searches is set at 30. This limit can be extended using this command. “limit” is an integer value. This command impacts the %path_check command, clock skew group definitions, and spice deck generation for logic cones. All three of these situations involve structural path or cone searches without the use of timing information.

(g) %set_rc_scale_factors [-min | -max] -res <res_factor> -cap <cap_factor> [-no_clk_rc_scaling]

- RC data (read from DSPF files) can be scaled up or down independently for max and min conditions, so that appropriate optimism or pessimism can be factored into the interconnect data easily. “res_factor” and “cap_factor” are floating-point numbers. Scale factors of 1.0 imply no change. Scale factors must be specified separately (in different commands) for min and max conditions. The “-no_clk_rc_scaling” switch disables RC scaling on clock nets.

(h) %set_delay_cal_pin_limit <limit>

- SDA’s default limit for number of pins per net is set at 50000. The above command can be used to change this limit. If any net has more pins than this limit, delay calculation will not be performed at that net and a warning will be issued to the log file.

(i) %disable_timing_anal

- Disable SDA’s default timing analysis while performing other functions that do not require timing information, such as static rule verification.

(j) %rpt_wire_load_model_usage

- Print a list of module instances in the design and the wire load models used within each instance.

(k) %set_new_escaped_name_rules

- Use standard SDF rules for interpreting escaped name strings in all command, DSPF and SDF input files. This essentially means that each escape character (“\”) affects only the immediately following character. If that following character normally has a special meaning -- such as a hierarchy divider (“/”) or a square bracket (“[“, “]”) -- the escape character forces it to be treated as an ordinary character. This command should be specified before all other commands.
- This treatment of escaped characters is very flexible, but it is different from standard Verilog rules, where all characters following an escape character are affected until the next white space. Verilog rules still apply to Verilog input files. Users should prepare the command, SDF and DSPF files in such a way that back-annotation will work correctly with escaped strings in Verilog.

(l) %ignore_dspf_divider

- Ignore the hierarchy divider character defined in DSPF files, and interpret all DSPF data as if the divider character is an ordinary character without special meaning. This command should only be used in conjunction with %set_new_escaped_name_rules.

(m) %ignore_sdf_divider

- Ignore the hierarchy divider character defined in SDF files, and interpret all SDF data as if the divider character is an ordinary character without special meaning. This command should only be used in conjunction with %set_new_escaped_name_rules.

11. APPLICATION PROGRAMMING INTERFACE (API)

The API is available for interfacing to SDA with other software systems such as floorplanners, logic synthesis tools and place-and-route tools. These systems can call SDA through a sequence of function calls to perform various timing and noise analysis tasks.

11.1 API Definition

The following "C" function calls constitute SDA's application programming interface. Note that many functions return a "boolean" value: if a TRUE (1) value is returned, then the function was successful; if a FALSE (0) value is returned, then the function resulted in an error. The "boolean" data type is a synonym for "int" in this document. Generic data types (such as "int", "long", "char", and "void") are used in defining the API as much as possible.

11.1.1 Data Input and Miscellaneous Functions

- **void sda_init()**
    - Initialize SDA. Must be issued at the beginning.
- **boolean sda_read_lib(char* libFileName, int min)**
    - Read a Synopsys library file. "libFileName" is the full path name for the file. Can be issued multiple times, to read multiple library files. If min_max = 1, then the library is a "min" library, otherwise it is a max or min/max library. Must be issued before reading netlist files. A FALSE return value indicates either a problem with opening the file, or an error, such as syntax error, while reading the file (details of the error are written to log file).
- **boolean sda_read_verilog(char* netlistFileName)**

- Read a Verilog netlist file. “netlistFileName” is the full path name for the file. Can be issued multiple times, to read multiple netlist files. Must be issued after reading library files and before creating database. A FALSE return value indicates either a problem with opening the file, or an error, such as syntax error, while reading the file (details of the error are written to log file).

- **boolean sda_create_db(char* topModuleName)**
  - Create internal database after reading all the library and netlist files. This function must be called after reading in the library and netlist data, but before reading in the user commands.

- **boolean sda_read_user_cmds(char* userCmdFileName)**
  - Read a user command file. “userCmdFileName” is the full path name for the file. Can be issued multiple times, to read multiple command files. Must be issued after creating database and before levelization. A FALSE return value indicates either a problem with opening the file, or an error, such as syntax error, while reading the file (details of the error are written to log file).

- **void sda_levelize()**
  - Levelize the internal database. Must be issued after reading all the command files, and before setting any RC and starting initial analysis.

- **boolean sda_assign_netID(char* hierNetName, long netID)**
  - Assign a unique “netID” (long integer; range: 0 to maxNets-1, where maxNets is the number of flat nets in the design) for the given hierarchical net name. The hierarchical name must be a string such as “/A/B/C/netname”. Must be issued for all nets in the design before specifying RC.

- **boolean sda_assign_cellID(char* hierCellName, long cellID)**
  - Assign a unique “cellID” (long integer; range: 0 to maxCells-1, where maxCells is the number of cell instances in the design) for the given hierarchical cell name. The hierarchical name must be a string such as “/A/B/C/cellname”. Must be issued for all cell instances in the design before specifying RC.

- **boolean sda_add_rc_to_net(long netID, double capVal)**
  - Specify a current net at which an RC network should be added. There can only be one “current net” at a time. The net is identified by its netID. “capVal” is the net’s total capacitance in fempto-farads. A FALSE return value indicates that the specified netID is invalid. This function should be issued first, followed by one or more instances (in any order) of the next

three functions to add resistors and capacitors. For incremental analysis, if the RC is to be changed at any net, these four functions should be issued again for that net.

- Note: Net names “GND” and “GND_dummy” are reserved for internal use, and should not be used as regular net names.

- **boolean sda_add_res(int nodeIndex1, long cellID1, char* pinName1, int nodeIndex2, long cellID2, char* pinName2, double resVal)**
  - Add a resistor between two sub-nodes within the RC network at the current net. Each sub-node is specified either as an index (for internal sub-nodes with the RC network), or as a combination of cellID and pin name for terminal sub-nodes that are connected to input/output pins of cell instances. Thus, if a valid nodeIndex is specified (any integer greater than 0), then cellID and pinName will be ignored. If nodeIndex is 0, then cellID and pinName will be used instead. For primary input or primary output nets, at least one sub-node should have nodeIndex set to -1, indicating a terminal connection to the I/O port. Primary input/output nets must be specified using the first three arguments only. This function can be issued multiple times to add multiple resistors. “resVal” is the resistance value in ohms.
- **boolean sda_add_cap(int nodeIndex, long cellID, char* pinName, double capVal)**
  - Add a grounded capacitor at a sub-node (specified according to the convention stated above) within the RC network at the current net. This function can be issued multiple times to add multiple capacitors. “capVal” is the capacitance value in ff (fempto-farads).
- **boolean sda_add_coupling(int nodeIndex1, long cellID1, char* pinName1, int nodeIndex2, long cellID2, char* pinName2, long net2ID, double capVal)**
  - Add a coupling capacitor between a sub-node (“1”) within the RC network at the current net and another sub-node (“2”) at a different net (specified according to the convention above). If nodeIndex2 is set to -2, the coupling capacitor will be connected to the special ground net “GND_dummy” (cellID2 and pinName2 will be ignored in that case). This function can be issued multiple times to add multiple coupling capacitors. “capVal” is the capacitance value in ff (fempto-farads). Note: Each coupling capacitor should only be specified at one of the nets that it connects to. SDA will then automatically duplicate it

at the other net in a way that allows for correct noise analysis. This convention is similar to the DSPF files expected by SDA.

- **void sda_duplicate_coupling()**
    - After all the RC networks at all nets have been specified, this function should be issued once in order to internally duplicate the coupling capacitors and to perform some other housekeeping tasks. This is crucial for proper noise analysis. For incremental analysis, this function should be issued after specifying new RC at some of the nets in the design.
    - Note: When explicit coupling is present between real nets (and not just between a real net and “GND_dummy”), if the RC is to be changed at any net for incremental analysis, the RC at all other nets connected to it via coupling must also be changed so that the entire net complex is consistent.
- **void sda_erase_all_rc()**
    - This function must be issued before replacing the RC data at all nets with new data for the next incremental analysis, so that the memory space used to store the RC data can be recovered and reused.
- **void sda_recycle_memory()**
    - This function must be issued after each reporting function in order to recover and reuse temporarily allocated memory.

11.1.2 Analysis and Verification Functions

- **void sda_initial_timing_analyze()**
    - Perform initial timing analysis, based on all the input data (including command files). If coupling analysis is enabled in the command files, then the NOD coupling analysis (noise-on-delay analysis for simultaneous switching) iterations will be automatically included in the analysis.
- **void sda_timing_verify()**
    - Perform setup/hold constraint verification and slack computation/propagation after initial or incremental timing analysis.
- **void sda_noise_analyze()**
    - Perform NOSN (noise-on-static-net) noise analysis throughout the design. Must be issued after completing at least the initial

timing analysis. Results will be stored at each victim net after the analysis. Can also be issued after an incremental timing analysis (only nets with changed parasitics or timing will be re-analyzed).

- **void sda_incremental_timing_analyze()**
    - Perform incremental timing analysis, based on updated parasitic data. (All other input data will remain unchanged.) This will automatically include NOD coupling analysis if enabled. Only portions of the design affected by parasitic or timing changes will be re-analyzed. This function is typically used for most incremental timing analyses, due to RC changes, during early and iterative phases of physical design. This function should not be used when RC data includes detailed coupling or when RC changes have occurred on clock nets or when the design uses transparent latches and relies on “cycle borrowing”.
- **void sda_clean_incremental_timing_analyze()**
    - Perform incremental timing analysis, based on updated parasitic data. (All other input data will remain unchanged.) This will automatically include NOD coupling analysis if enabled. This function must be used for incremental timing analyses, due to RC changes, when detailed coupling data is present or when RC changes have occurred on clock nets or when the design contains transparent latches and relies on “cycle borrowing”.
- **void sda_allocate_negative_slack()**
    - Allocate negative slack of each path to the nets along the path, proportionate to the pin-to-pin RC delays of nets, for supporting the router’s “delay improvement” step. If a net is part of multiple paths, the worst-case slack and path will be used for allocating negative slack to that net. Negative slack will be allocated to each pin-to-pin arc (from driver pin to receiver pin) on each net, and stored at the receiver pin. This function should be issued once after each complete (initial or incremental) timing analysis/verification iteration.

11.1.3 Output and Reporting Functions

- **void* sda_get_violating_nets(int minSlack, unsigned short opCond)**
    - Obtain a list of nets in the design that have slacks less than the specified minSlack (units: pico-seconds). opCond is 0 max

(worst-case) and 1 for min (best-case). The function returns a pointer which points to a singly linked list. Each element on the list refers to one net, and contains two pointers: the first pointer points to long integer containing the netID, and the second pointer points to an integer containing the worst-case slack. (SDA's "sda_doublePtrList" data structure will be used to construct the linked list[9].)

- **void* sda_get_timing_report(int minSlack, unsigned short opCond)**
    - Obtain a list of critical paths that have slacks less than the specified minSlack (units: pico-seconds). The return pointer points to a singly linked list (implemented using SDA's "sda_triplePtrList" data structure). Each element on the list represents a single critical path, and contains two pointers: the first pointer points to an integer containing the slack for the path, the second pointer points to another singly linked list (implemented using SDA's "sda_triplePtrList" data structure[10]) to represent that entire critical path, and the third pointer points to an integer whose value indicates whether the path has a rising (1) or falling (0) edge at the end point.
    - Within the critical path, each element represents a net/pin on the path, and contains three pointers: the first pointer points to a long integer containing a cellID (or -1 if it is a primary input/output net), the second pointer points to a character string containing a pin name at the specified cell instance (or net name of primary input/output), and the third pointer points to an integer containing the arrival time (units: pico-seconds) at that pin.
- **void* sda_get_sensitivity_matrix(int type, void* netIdList)**
    - Obtain a sensitivity matrix for either NOD or NOSN (type = 0 for NOD and type = 1 for NOSN). "netIdList" is a singly linked list (implemented using SDA's "sda_singlePtrList" data structure), with the pointer at each element pointing to a long integer containing a netID. The sensitivity matrix is then constructed by setting element i-j (for net at position i of netIdList as victim, and net at position j as aggressor) equal to 1 when the timing windows indicate possible NOD or NOSN impact, and 0 if no impact.
    - The function returns a pointer to the matrix structure. The sensitivity matrix is a singly linked list (implemented using

[9] Please refer to the file "lists.h" in the "include" directory for details of SDA's list data structures.
[10] The newly introduced sda_triplePtrList is similar to sda_doublePtrList, except that each list element includes three pointers.

SDA's "sda_singlePtrList" data structure), with each element (a single pointer) pointing to one row of the matrix. The pointer points to a singly linked list representing the entire row, with each element of that row list (a pointer) pointing to an integer containing the value of the matrix at that i-j position.

- **void* sda_get_negative_slack(long netID)**
    - Obtain the allocated negative slack for a net. This function should be issued only after "sda_allocate_negative_slack" has been issued. The function returns a pointer to a singly linked list (implemented using SDA's new "sda_triplePtrList" data structure). Each element on the list contains three pointers: The first pointer points to a long integer containing a cellID whose pin is connected to the net (or -1 for primary output), the second pointer points to a character string containing the pin name (or net name for primary output), and the third pointer points to an integer containing the negative slack for that pin on the net (normally, this negative slack is <= 0; or 100000000 if no negative slack path passes through the net).
- **boolean sda_get_noise_info(long victimNetID)**
    - Obtain the latest NOSN noise information at a specific victim net. The "edge" argument specifies whether rising or falling noise must be considered (0 = rise, 1 = fall). The returned value is TRUE if there is a noise violation at this net (i.e., noise pulse exceeds the threshold specified through the command file), FALSE otherwise. This function must be called after sda_noise_analyze(). It considers the composite noise from all time-aligned aggressors.
- **int sda_get_slew_info(long cellID, char* pinName, unsigned short edge)**
    - Obtain the slew rate (slope) at the specified pin of the specified cell instance (unit: pico-seconds). The slope is measured between the same two thresholds as elsewhere in the timing analysis and reporting.
- **int sda_get_cell_delay(long cellID, char* inputPinName, char* outputPinName, unsigned short edge)**
    - Obtain the maximum gate delay from an input pin to an output pin of the specified cell instance (unit: pico-seconds). "edge" is 1 for rising and 0 for falling.
- **int sda_get_net_delay(long drvgCellID, char* drvgPinName, long rcvgCellID, char* rcvgPinName, unsigned short edge)**

- Obtain the maximum net delay from a driving pin to a receiving pin at the specified net (unit: pico-seconds). “edge” is 1 for rising and 0 for falling.

- **int sda_get_net_drive_resistance(long netID, unsigned short edge)**
  - Obtain a numerical value for the drive resistance (unit: ohms) of a net. “edge” is 1 for rising and 0 for falling. The drive resistance will be calculated as the sum of the gate output resistance and the wire resistance. Both of these items will be estimated approximately based on the available information and then added together. (The drive strength of the net will be inversely proportional to the drive resistance.)

- **int sda_get_gate_capacitance(char* cellType, char* pinName)**
  - Obtain the gate capacitance (units: fempto-farads) for a specified pin of a specified cell type in the library.

- **int sda_get_net_ceff(long netID)**
  - Obtain the effective net capacitance (units: fempto-farads), based on SDA’s internal delay calculation.

- **int sda_get_net_reff(long netID)**
  - Obtain the effective net resistance (units: ohms), based on SDA’s internal delay calculation.

- **boolean sda_output_log(char* logFileName)**
  - Specify full path name to the output log file. SDA will output details of any warnings or fatal errors to this file (similar to the file specified by the “-log” option in batch-mode). A FALSE return value indicates that the specified file could not be opened for writing. In normal operation, the router will not examine the contents of this file; however, in case of any fatal error (as indicated by the return boolean value of other API functions), contents of this log file may be useful for further off-line debugging.

- **boolean sda_output_rpt(char* rptFileName)**
  - Specify full path name to the output report file. This is an optional function that can be used if a standard file-based timing or noise report is needed. It is generally necessary to define such an output file, even if some of the results are being accessed directly through other API functions, particularly when NOSN noise analysis is being done.

- **void sda_timing_report()**

- Output the timing report to the previously specified report file. (The NOSN noise report is written automatically to the report file.)

- **void sda_dspf_report()**
  - Output the dspf data to the file “rptFileName”.dspf.

11.1.4 Support Functions for Distributed Analysis

- **boolean sda_partition_design(int minSize, int maxSize, char* outFileName)**
  - Perform logic partitioning without timing analysis. Equivalent of the batch-mode command “%enable_logic_partitioning -optimal 6 -no_timing -max_size M -min_size N -preserve_names”.
- **boolean sda_generate_timing_model(char* topModuleName, char* outFileName)**
  - Generate a timing model for the current design. Equivalent of the batch-mode command “%io_timing_model_gen”. The timing model will be written to the file “outFileName.lib”. Note: The min/max condition for the model will be based on the %set_condition specification in the command file.
- **void sda_prepare_to_generate_io_constraints()**
  - This function must be called at the beginning of any program that will later call the following function to generate I/O timing constraints.
- **boolean sda_generate_block_io_constraints()**
  - Generate I/O timing constraints for top-level black-boxed module instances. Equivalent of the batch-mode command “%io_constr_gen”.
- **boolean sda_export_timing(unsigned short cond, long netId, int* riseArrTime, unsigned short* riseClkId, unsigned short* riseLtFlag, int* fallArrTime, unsigned short* fallClkId, unsigned short* fallLtFlag, int* riseDrvgptSlope, int* fallDrvgptSlope)**
  - Timing information at a local net (for example, when a block is being analyzed in detail) can be extracted (exported) by an external master process through this function call and stored in a global register. Designed to support a distributed timing/noise analysis methodology. Should be issued after a min0, max2 or maxp (max’) single-condition coupling analysis within a block.

- cond = 1 (min0) or 0 (maxp or max2). (maxp and max2 are differentiated only by the analysis done based on the "%set_condition" command.)
- rise and fall driving point slopes are meaningful only when the condition is maxp.

- **boolean sda_import_timing(unsigned short cond, long netId, int riseArrTime, unsigned short riseClkId, unsigned short riseLtFlag,int fallArrTime, unsigned short fallClkId, unsigned short fallLtFlag, int riseDrvgptSlope, int fallDrvgptSlope)**
  - Timing information at a local net (or global aggressor net -- see below) can be specified (imported) by an external master process through this function call. Designed to support a distributed timing/noise analysis methodology. Should be issued before an NOD or NOSN analysis within a block. Min0 and max2 timing information must be imported for global aggressor nets in an NOD analysis, and maxp (max') timing information must be imported for all nets in an NOSN analysis.
  - cond = 1 (min0), 0 (maxp), 2 (max2).
  - Rise and fall driving point slopes are meaningful only when the condition is maxp.
- **boolean sda_set_max_netID(long maxNetID)**
  - Specify the maximum global value of net IDs (starting from 0) to a local block process when an external master process is managing multiple blocks in a distributed analysis methodology. This value should be greater than the number of nets within the block. Must be issued after levelization.
- **boolean sda_set_max_cellID(long maxCellID)**
  - Specify the maximum global value of cell IDs (starting from 0) to a local block process when an external master process is managing multiple blocks in a distributed analysis methodology. This value should be greater than the number of cell instances within the block. Must be issued after levelization.
- **boolean sda_add_global_aggressor_net(long netID)**
  - An external master process can define specific global aggressor nets within a block process (these nets are outside of the block netlist) in a distributed NOD/NOSN coupling analysis methodology. Must be issued after setting max net/cell IDs, and before RC setting and analysis.
- **boolean sda_add_global_aggressor_cell_pin (long netID, long cellID, char* libCellName, char* pinName)**

- Must be issued (one or more times, for each cell pin on a global aggressor net) after adding a netID in the previous function and before RC setting or analysis.

- **boolean sda_add_global_aggressor_pio_pin(long netID, int pinType)**
  - Must be issued once (for any primary input/output port on a global aggressor net) after adding a netID in the previous function and before RC setting or analysis.
  - pinType is 0 for primary input, and 1 for primary output.

11.2 API Usage

All of the SDA API functions start with the prefix "sda_". This prefix is reserved for SDA and should not be used by other software systems that use this API.

The API functions must be called in certain specific sequences in order to obtain meaningful and predictable results, as illustrated below. Also, any function that has the "boolean" return type indicates that error status is being returned. In such cases, the return value must be checked to make sure that it is "TRUE" before proceeding to the next step. "TRUE" indicates that the operation was successful. "FALSE" indicates a fatal error. The most common fatal error conditions arise due to specification of incorrect name strings, such as file names, module names, hierarchical net/pin names, etc.

Most of the design data and constraints (Verilog netlist files, library files and command files) are read in as directed by the appropriate API functions. Parasitic RC data is passed via API functions on a net-by-net basis, which allows the RC data to be incrementally modified. Whenever the RC data is modified, it is possible to run an incremental timing or noise analysis.

APPENDIX: A-1. EXAMPLES OF API USAGE

The following sample "C" programs illustrate how the SDA API is correctly used for various types of timing and noise analysis operations.

A-1.1 Basic Timing Analysis and Reporting

```
/* Include system header  files */
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <stdlib.h>
```

```
/* Include SDA definitions/mnemonics */

#include "defines.h"

/* Include SDA header files needed for globals.h */

#include "lists.h"
#include "lib.h"
#include "prim.h"
#include "stage.h"
#include "sdf.h"
#include "hnli.h"
#include "hdb.h"
#include "ui.h"
#include "globals.h"

#include "ioApi.ch"
#include "analApi.ch"
#include "rptApi.ch"
#include "lists.ch"

main(int argc, char *argv[])
{
  singlePtrListS* listP;
  singlePtrListS* rlistP;
  singlePtrListS* slistP;
  singlePtrListS* nlistP;
  doublePtrListS* rptP;
  doublePtrListS* dlistP;
  triplePtrListS* trptP;
  triplePtrListS* trlistP;
  triplePtrListS* tlistP;
  long* longPtr;
  int* intPtr;
  int* edgePtr;
  char* namePtr;

  /* load design */

  sda_init();
  if (!sda_output_log("log"))
          exit(0);
  if (!sda_output_rpt("test2.out"))
          exit(0);
  if (!sda_read_lib("equ_syn.lib", 0))
          exit(0);
  if (!sda_read_verilog("test2.v"))
          exit(0);
  if (!sda_create_db("test2"))
          exit(0);
  if (!sda_read_user_cmds("test2.cmd"))
          exit(0);
  sda_levelize();
```



```
/* read dspf net by net */

sda_assign_netID("/n2", 0);
sda_assign_netID("/a1", 1);
sda_assign_netID("/a1n", 2);
sda_assign_netID("/a1nn", 3);
sda_assign_netID("/n1", 4);
sda_assign_netID("/b1", 5);
sda_assign_netID("/n2x", 6);
sda_assign_netID("/n2xb", 7);
sda_assign_netID("/n2xbb", 8);
sda_assign_netID("/n2xbbb", 9);
sda_assign_netID("/n3", 10);
sda_assign_netID("/y1", 11);
sda_assign_netID("/n4", 12);
sda_assign_netID("/n5", 13);
sda_assign_netID("/z1", 14);
sda_assign_netID("/d", 15);
sda_assign_netID("/c", 16);

sda_assign_cellID("/nd2x2", 0);
sda_assign_cellID("/lx", 1);
sda_assign_cellID("/ix4", 2);
sda_assign_cellID("/ixx1", 3);
sda_assign_cellID("/ixx2", 4);
sda_assign_cellID("/dffx1", 5);
sda_assign_cellID("/nd2x1", 6);
sda_assign_cellID("/ix3", 7);
sda_assign_cellID("/ix3a", 8);
sda_assign_cellID("/ix3b", 9);
sda_assign_cellID("/ix", 10);
sda_assign_cellID("/nd2x3", 11);
sda_assign_cellID("/ix2", 12);

/* Net n2 */
if (!sda_add_rc_to_net(0, 3219))
        exit(0);
sda_add_cap(0, 0, "b", 114);
sda_add_cap(2, 0, NULL, 1238);
sda_add_cap(3, 0, NULL, 21);
sda_add_cap(4, 0, NULL, 28);
sda_add_cap(5, 0, NULL, 7);
sda_add_cap(6, 0, NULL, 1048);
sda_add_cap(0, 1, "d", 367.8);
sda_add_cap(8, 0, NULL, 200);
sda_add_cap(9, 0, NULL, 7);
sda_add_cap(10, 0, NULL, 188.4);

sda_add_res(0, 2, "y", 0, 0, "b", 10);
sda_add_res(0, 0, "b", 2, 0, NULL, 72);
sda_add_res(2, 0, NULL, 3, 0, NULL, 34);
sda_add_res(3, 0, NULL, 4, 0, NULL, 96);
sda_add_res(4, 0, NULL, 5, 0, NULL, 72);
sda_add_res(4, 0, NULL, 2, 0, NULL, 100);
sda_add_res(5, 0, NULL, 6, 0, NULL, 10);
```



```
sda_add_res(6, 0, NULL, 0, 1, "d", 120);
sda_add_res(0, 1, "d", 8, 0, NULL, 24);
sda_add_res(0, 0, "b", 9, 0, NULL, 48);
sda_add_res(9, 0, NULL, 10, 0, NULL, 24);


/* analyze and report */
sda_initial_timing_analyze();
sda_timing_verify();

  /* violating nets */

fprintf(stdout, "\n\n");
fprintf(stdout, "Violating Nets -- Slack Report\n\n");
rptP = sda_get_violating_nets(-100, 0);

listIterate(dlistP, rptP)
{
        longPtr = sda_getDptr1(dlistP);
        intPtr = sda_getDptr2(dlistP);
        fprintf(stdout, "NetId = %d          :          Slack = %d\n", *longPtr, *intPtr);
}

sda_recycle_memory();

/* critical paths */

fprintf(stdout, "\n\n");
fprintf(stdout, "Critical Path Report\n\n");

trptP = sda_get_timing_report(-1000, 0);

listIterate(trlistP, trptP)
{
        intPtr = sda_getTptr1(trlistP);
        edgePtr = sda_getTptr3(trlistP);

        if (*edgePtr == 1)
                fprintf(stdout, "Slack = %d          Edge = RISE\n", *intPtr);
        else
                fprintf(stdout, "Slack = %d     Edge = FALL\n", *intPtr);

        listIterate(tlistP, sda_getTptr2(trlistP))
        {
                longPtr = sda_getTptr1(tlistP);
                namePtr = sda_getTptr2(tlistP);
                intPtr = sda_getTptr3(tlistP);
                fprintf(stdout, "CellId = %d          pin = %s          arr = %d\n",
                        *longPtr, namePtr, *intPtr);
        }

        fprintf(stdout, "\n");
}

sda_recycle_memory();
```

```
}
```

A-1.2 Basic Noise Analysis and Reporting

```
/* Include system header  files */
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <unistd.h>

/* Include SDA definitions/mnemonics */

#include "defines.h"

/* Include SDA header files needed for globals.h */

#include "lists.h"
#include "lib.h"
#include "prim.h"
#include "stage.h"
#include "sdf.h"
#include "hnli.h"
#include "hdb.h"
#include "ui.h"
#include "globals.h"

#include "ioApi.ch"
#include "analApi.ch"

main(int argc, char *argv[])
{
  /* load design */

  sda_init();
  if (!sda_output_log("log"))
          exit(0);
  if (!sda_output_rpt("nosn_test_accmode.out"))
          exit(0);
  if (!sda_read_lib("equ_syn_nosn_test.lib", 0))
          exit(0);
  if (!sda_read_verilog("nosn_test.v"))
          exit(0);
  if (!sda_create_db("nosn_test"))
          exit(0);
  if (!sda_read_user_cmds("nosn_test_accmode.cmd"))
          exit(0);
  sda_levelize();

  /* read dspf net by net */

  sda_assign_netID("/net1", 0);
  sda_assign_netID("/net2", 1);
```

```
  sda_assign_cellID("/inv1", 0);
  sda_assign_cellID("/inv2", 1);
  sda_assign_cellID("/inv3", 2);
  sda_assign_cellID("/inv4", 3);

  /* Net net1 */
  if (!sda_add_rc_to_net(0, 1000))
          exit(0);
  sda_add_res(0, 0, "y", 0, 2, "a", 1000);
  sda_add_cap(0, 2, "a", 500);
  sda_add_cap(0, 0, "y", 1);
  sda_add_coupling(0, 2, "a", 0, 3, "a", 1, 1000);

  /* Net n2 */
  if (!sda_add_rc_to_net(1, 1000))
          exit(0);
  sda_add_res(0, 1, "y", 0, 3, "a", 1000);
  sda_add_cap(0, 3, "a", 1000);
  sda_add_cap(0, 1, "y", 1);

  sda_duplicate_coupling();

  /* analyze and report */
  sda_initial_timing_analyze();
  sda_timing_verify();
  sda_noise_analyze();
  sda_timing_report();
}
```

A-1.3 Incremental Timing Analysis after RC Changes

```
/* Include system header  files */
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <unistd.h>

/* Include SDA definitions/mnemonics */

#include "defines.h"

/* Include SDA header files needed for globals.h */

#include "lists.h"
#include "lib.h"
#include "prim.h"
#include "stage.h"
#include "sdf.h"
#include "hnli.h"
#include "hdb.h"
#include "ui.h"
#include "globals.h"
```



```
#include "ioApi.ch"
#include "analApi.ch"

main(int argc, char *argv[])
{
  /* load design */

  sda_init();
  if (!sda_output_log("log"))
          exit(0);
  if (!sda_output_rpt("test_cell.out"))
          exit(0);
  if (!sda_read_lib("equ_syn.lib", 0))
          exit(0);
  if (!sda_read_verilog("test_cell.v"))
          exit(0);
  if (!sda_create_db("test_cell"))
          exit(0);
  if (!sda_read_user_cmds("test_cell.cmd"))
          exit(0);
  sda_levelize();

  /* read dspf net by net */

  sda_assign_netID("/a1", 0);
  sda_assign_netID("/n2", 1);
  sda_assign_netID("/n5", 2);

  sda_assign_cellID("/ixx1", 0);
  sda_assign_cellID("/nd2x1", 1);
  sda_assign_cellID("/lx", 2);
  sda_assign_cellID("/nd2x2", 3);
  sda_assign_cellID("/nd2x3", 4);
  sda_assign_cellID("/ix2", 5);

  /* Net a1 */
  if (!sda_add_rc_to_net(0, 200))
          exit(0);
  sda_add_cap(-1, 0, NULL, 100);
  sda_add_cap(1, 0, NULL, 100);
  sda_add_res(-1, 0, NULL, 1, 0, NULL, 100);
  sda_add_res(1, 0, NULL, 0, 0, "a", 100);

  /* Net n2 */
  if (!sda_add_rc_to_net(1, 300))
          exit(0);
  sda_add_cap(0, 1, "y", 100);
  sda_add_cap(0, 2, "d", 50);
  sda_add_cap(0, 3, "b", 50);
  sda_add_cap(1, 0, NULL, 100);
  sda_add_res(0, 1, "y", 1, 0, NULL, 100);
  sda_add_res(1, 0, NULL, 0, 2, "d", 100);
  sda_add_res(1, 0, NULL, 0, 3, "b", 100);
```



```
  /* Net n5 */
  if (!sda_add_rc_to_net(2, 100))
          exit(0);
  sda_add_cap(0, 4, "y", 25);
  sda_add_cap(0, 5, "a", 75);
  sda_add_res( 0, 4, "y", 0, 5, "a", 100);

  /* initial analysis and verification */
  sda_initial_timing_analyze();
  sda_timing_verify();

  /* incremental RC change (new RC net by net) */
  /* Net n2 */
  if (!sda_add_rc_to_net(1, 300))
      exit(0);
  sda_add_cap(0, 1, "y", 100);
  sda_add_cap(0, 2, "d", 500);
  sda_add_cap(0, 3, "b", 400);
  sda_add_cap(1, 0, NULL, 100);
  sda_add_res(0, 1, "y", 1, 0, NULL, 100);
  sda_add_res(1, 0, NULL, 0, 2, "d", 300);
  sda_add_res(1, 0, NULL, 0, 3, "b", 200);

  /* final analysis, verification and report */
  sda_incremental_timing_analyze();
  sda_timing_verify();
  sda_timing_report();
}
```


A-1.4 Incremental Noise Analysis after RC Changes

```
/* Include system header  files */
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <unistd.h>

/* Include SDA definitions/mnemonics */

#include "defines.h"

/* Include SDA header files needed for globals.h */

#include "lists.h"
#include "lib.h"
#include "prim.h"
#include "stage.h"
#include "sdf.h"
#include "hnli.h"
#include "hdb.h"
#include "ui.h"
#include "globals.h"

#include "ioApi.ch"
#include "analApi.ch"

main(int argc, char *argv[])
{
  /* load design */

  sda_init();
  if (!sda_output_log("log"))
          exit(0);
  if (!sda_output_rpt("nosn_test_accmode.out"))
          exit(0);
  if (!sda_read_lib("equ_syn_nosn_test.lib", 0))
          exit(0);
  if (!sda_read_verilog("nosn_test.v"))
          exit(0);
  if (!sda_create_db("nosn_test"))
          exit(0);
  if (!sda_read_user_cmds("nosn_test_accmode.cmd"))
          exit(0);
  sda_levelize();

  /* read dspf net by net */

  sda_assign_netID("/net1", 0);
  sda_assign_netID("/net2", 1);

  sda_assign_cellID("/inv1", 0);
```



```
  sda_assign_cellID("/inv2", 1);
  sda_assign_cellID("/inv3", 2);
  sda_assign_cellID("/inv4", 3);

  /* Net net1 */
  if (!sda_add_rc_to_net(0, 1000))
          exit(0);
  sda_add_res(0, 0, "y", 0, 2, "a", 1000);
  sda_add_cap(0, 2, "a", 500);
  sda_add_cap(0, 0, "y", 1);
  sda_add_coupling(0, 2, "a", 0, 3, "a", 1, 1000);

  /* Net n2 */
  if (!sda_add_rc_to_net(1, 1000))
          exit(0);
  sda_add_res(0, 1, "y", 0, 3, "a", 1000);
  sda_add_cap(0, 3, "a", 1000);
  sda_add_cap(0, 1, "y", 1);

  sda_duplicate_coupling();

  /* initial analyze and report */
  sda_initial_timing_analyze();
  sda_timing_verify();
  sda_noise_analyze();

  /* incremental RC change for net complex */
  sda_erase_all_rc();

  /* Net net1 */
  if (!sda_add_rc_to_net(0, 1000))
      exit(0);
  sda_add_res(0, 0, "y", 0, 2, "a", 500);
  sda_add_cap(0, 2, "a", 200);
  sda_add_cap(0, 0, "y", 1);
  sda_add_coupling(0, 2, "a", 0, 3, "a", 1, 1500);

  /* Net n2 */
  if (!sda_add_rc_to_net(1, 1000))
      exit(0);
  sda_add_res(0, 1, "y", 0, 3, "a", 500);
  sda_add_cap(0, 3, "a", 500);
  sda_add_cap(0, 1, "y", 1);

  sda_duplicate_coupling();

  /* final analysis, verification and report */
  sda_clean_incremental_timing_analyze();
  sda_timing_verify();
  sda_noise_analyze();
  sda_timing_report();
}
```

APPENDIX: A-2. EXAMPLES OF STATIC CONSTRAINT VERIFICATION

This section provides several examples that illustrate how *DesignCheck* can be used in typical verification applications. Most of these examples have been modeled after actual usage of the tool by designers in various applications. In each case, the rule is first stated in English, followed by a formal *DesignCheck* statement which captures the rule.

A-2.1 Examples of Cell Usage Verification

- Verify that there are no floating inputs at any cell instance in the entire design:

  %cell_check -all -prop { no_flt_input }

- Verify that there are no floating inputs at all instances of the cell type nd2:

  %cell_check -cell nd2 -prop { no_flt_input }

- Verify that there are no floating inputs at cell instance lx1:

  %cell_check -cell -inst /lx1 -prop { no_flt_input }

- Verify that input pins S and SN of multiplexer cell type mx2 are always driven by logically complementary signals:

  %cell_check -cell mx2 -prop { inv } -pin1 S -pin2 SN

- Verify that input pins S and SN of cell instance mx2x8 are driven by logically equivalent signals:

  %cell_check -cell -inst /mx2x8 -prop { equ } -pin1 S -pin2 SN

- Verify that input pins S and SN of cell instance mx2x8 are driven by mutually exclusive signals:

  %cell_check -cell -inst /mx2x8 -prop { mut_excl } -pin1 S -pin2 SN

- Verify that input pins S0, S1 and S2 of cell instance mx4x8 are driven by mutually exclusive signals:

  %cell_check -cell -inst /mx2x8 -prop { mut_excl } -pinlist {S0 S1 S2}

- Verify that input pins A and B of cell type nd2 are always driven by opposite-phase signals:

%cell_check -cell nd2 -prop { opp_phase } -pin1 A -pin2 B

- Verify that input pins A and B of cell instance nd2x8 are driven by same-phase signals:

  %cell_check -cell -inst /nd2x8 -prop { same_phase } -pin1 A -pin2 B

- Verify that the cell type imux4 does not drive a tristate or multi_drive net:

  %cell_check -cell imux4 -prop { invalid -net_type tristate } -output
  %cell_check -cell imux4 -prop { invalid -net_type multi_drive } -output

- Verify that the cell type tsbuf always drives a tristate net:

  %cell_check -cell tsbuf -prop { valid -net_type tristate } -output

- Verify that pin Z of cell type nr5 does not drive more than 3 receivers:

  %cell_check -cell nr5 -prop { max_fanout 3 } -output -pin Z

A-2.2 Examples of Module Level Verification

- Verify that the module test_cell instantiates only two types of sequential cells, dff and latch:

  %module_check -module test_cell -prop { valid seq } -cell_list { dff latch}

- Verify that the module test_cell does not instantiate the cell types dffts and any cell type that matches the string l*h:

  *%module_check -module test_cell -prop { invalid } -cell_list { dffts l*h }*

A-2.3 Examples of Functional Verification

- Verify that all the tristate drivers on the net Dout are enabled by mutually exclusive signals:

  func_check -prop { valid mut_excl_drivers } -net Dout

- Verify that the tristate drivers on *all tristate nets* are enabled by mutually exclusive signals:

  %func_check -prop { valid mut_excl_drivers } -net_type tristate

- Verify that all tristate nets have either default drivers or are driven by weak holder cells of the type whld:

  %set_cell_prop -weak_holder -cell whld
  %func_check -prop { valid default_driver } -net_type tristate

- Verify that two given hierarchical nets have the "scpc" property (i.e., they do not switch in opposite directions simultaneously):

  %func_check -prop { valid scpc } -net a n1

- Verify that two given hierarchical nets have the "scnc" property (i.e., they may switch in opposite directions simultaneously):

  %func_check -prop { valid scnc } -net nc1 n2

- Verify that two given hierarchical nets have the "dcc" property (i.e., they do not switch in the same clock cycle):

  %func_check -prop { valid dcc } -net ts1 ts2a

- Verify that two given hierarchical nets have the "scahc" property (i.e., one net is in an active high state while the other, presumably a tristate net, is being actively driven):

  %func_check -prop { valid scahc } -net na3b ts2a

  Note that this type of constraint can be used to verify that the value on a tristate net is "read" or latched only when the net is being actively driven. The other net in the rule could be a latch enable or other sampling signal within the receiving logic.

A-2.4 Examples of Timing Correlation Verification

- Determine whether there is any overlap of switching (timing) windows at the two hierarchical nets specified in the following command:

  %coupling_check /pqr/n1 /z1

A-2.5 Examples of General Timing Constraint Verification[11]

This section illustrates the use of some of the SDF-based general timing constraint specifications.

[11] These constraints are specified in an SDF file.

- Verify that all instances of the cell type nd2 meet the specified maximum skew constraints between various edges at the inputs A and B:

```
(CELL
    (CELLTYPE "nd2")
    (INSTANCE * )
    (TIMINGCHECK
            (SKEW (posedge A) (posedge B) (700))
            (SKEW (negedge A) (posedge B) (290))
    )
)
```

- Verify that the specified global paths meet the maximum rise and fall delay constraints:

```
(CELL
    (CELLTYPE "top")
    (INSTANCE )
    (TIMINGENV
            (PATHCONSTRAINT path1 pqr.nd2x1.A pqr.ixx.Z nd2x2.A (2005) (2000))
            (PATHCONSTRAINT path2 pqr.nd2x1.A pqr.ixx.Z nd2x1.Z (2005) (2000))
            (PATHCONSTRAINT path3 pqr.nd2x1.A nd2x2.A y1 (2405) (2400))
    )
)
```

- Verify that the spread of delays from the output of the specified cell instance to all its receivers is within the maximum skew constraint:

```
(CELL
    (CELLTYPE "nd2")
    (INSTANCE pqr.nd2x1)
    ( TIMINGENV
            ( SKEWCONSTRAINT Z (100) )
            ( SKEWCONSTRAINT (negedge Z) (100) )
            ( SKEWCONSTRAINT (posedge Z) (100) )
    )
 )
```

- Verify the specified recovery and removal constraints on asynchronous preset and clear inputs of a flip-flop instance, as well as the normal setup and hold constraints for the synchronous data input:

```
(CELL
    (CELLTYPE "dffpc")
    (INSTANCE ff1)
    (TIMINGCHECK
            (SETUP D (posedge CLK) (230:230:230) )
            (HOLD D (posedge CLK) (130::130) )
            (RECOVERY (negedge PRT) (posedge CLK) (200) )
            (REMOVAL (negedge PRT) (posedge CLK) (100) )
```

```
            (RECOVERY (negedge CLR) (posedge CLK) (250) )
            (REMOVAL (negedge CLR) (posedge CLK) (150) )
    )
  )
```

- Verify the DIFF and LESS constraints between various global paths:

```
(CELL
    (CELLTYPE "top")
    (INSTANCE )
    (TIMINGENV
            (DIFF
                    (pqr.nd2x1.A pqr.ixx.Z nd2x2.Z)
                    (pqr.nd2x1.A pqr.ixx.A nd2x1.A)
                    (100) (100)
            )
            (DIFF
                    (pqr.nd2x1.A pqr.ixx.A nd2x1.A)
                    (pqr.nd2x1.A pqr.ixx.Z nd2x2.Z)
                    (100) (100)
            )
            (LESS
                    (pqr.nd2x1.A pqr.ixx.A nd2x1.A)
                    (pqr.nd2x1.A pqr.ixx.Z nd2x2.Z)
                    (0) (100)
            )
            (LESS
                    (pqr.nd2x1.A pqr.ixx.Z nd2x2.Z)
                    (pqr.nd2x1.A pqr.ixx.A nd2x1.A)
                    (0) (100)
            )
    )
)
```

A-2.6 Examples of Logic Value Based Verification

- Set logic high values at nets n1 and b1, propagate them through combinational logic, and then verify that nets n2 and n4c have the required logic values as a result:

```
%set_logic_value -H /n1
%set_logic_value -H /b1
%prop_logic_values
%logic_value_check -H n2
%logic_value_check -L n4c
```

A-2.7 Examples of False Path Verification

- Verify whether the given path (ordered, complete sequence of nodes) is a functionally invalid or false path:

%false_path_check /dffx1/Q /x6/A /x6/Z /x7/A /x8/A /x8/Z /x9/A /dffx2/D

This test can be applied to critical paths generated by a timing analyzer. The sequence of cell inputs is used in the verification to uniquely identify the path, and any cell output in the path is ignored.

A-2.8 Examples of Path Validity Verification

- Verify that a particular register file cell (regfile) has its write-data (WD) and write-address (WA) pins driven only from two specific types of latch cells (latch1, latch2):

  %path_check -prop { excl_dest } -src -cell latch1
  -src -cell latch2
  -dest -cell regfile -pin WD
  -dest -cell regfile -pin WA

  Note that the rule is written in terms of cell types, and not instances. This covers all instances of the regfile cell.

- Verify that the register file cell (regfile) drives exactly one instance of a special buffer cell (regbuf), but may drive other fanout without any restriction:

  %path_check -prop { valid } -src -cell regfile -dest -cell regbuf -num 1

- Verify that the paths between any two precharge cells are always inverting. pcell1 and pcell2 are the two types of precharge cells available in this case, and static inverters are to be instantiated outside of the precharge cells:

  %path_check -prop { valid inv } -src pcell1 -src pcell2
  -dest pcell1 -dest pcell2

- Verify that the paths between two cell types (cell1 to cell2) are always non-inverting:

  %path_check -prop { valid ninv } -src cell1 -dest cell2

- Verify that any path ending at a latch (any latch in the design) does not originate from a latch or flip-flop which is clocked by the same phase as the destination latch:

  %path_check -prop { invalid same_phase } -src -latch -src -ff -dest -latch

Note that generic "-latch" and "-ff" keywords are used in this statement to cover *all* latches or flip-flops in the design.

- Verify that there are no "same-phase" paths ending at any latch, except if the paths originate from two types of precharge cells (prech1, prech2):

```
%path_check   -prop { invalid same_phase }   -src -latch
                                             -src -ff
                                             -except -src prech1
                                             -except -src prech2
                                             -dest -latch
```

- Verify that all clock pins of flip-flops and latches are driven from one of two clock nets (phi1, phi2) in a module:

```
%path_check   -prop { excl_dest }   -src -net phi1
                                    -src -net phi2
                                    -dest -ff -pin CLK
                                    -dest -ff -pin CK
                                    -dest -ff -pin WE
                                    -dest -latch -pin CLK
                                    -dest -latch -pin CK
```

Note that generic flip-flops ("-ff") and latches ("-latch") are called out multiple times among the destinations, so that multiple clock pin names can be covered. For example, there are three possible clock pin names (CLK, CK, WE) that cover all possible flip-flop cells in this design.

- Verify that a clock input to a design (net CLK) drives exactly one instance of a clock gate cell (clk_gate). Further, verify that the clk_gate cell drives only the clock pins of latches and flip-flops, possibly through some intermediate buffering but without any net inversion. Together, these statements verify that there is exactly one level of clock gating in the design.

```
%path_check   -prop { excl_src }    -src -net CLK
                                    -dest -cell clk_gate -num 1
%path_check   -prop { excl_src }    -src -cell clk_gate
                                    -dest -ff -pin CLK
                                    -dest -ff -pin CK
                                    -dest -ff -pin WE
                                    -dest -latch -pin CLK
                                    -dest -latch -pin CK
%path_check   -prop { excl_dest }   -src -cell clk_gate
                                    -dest -ff -pin CLK
                                    -dest -ff -pin CK
```


-dest -ff -pin WE
-dest -latch -pin CLK
-dest -latch -pin CK

- Verify that specified inputs (pins X and Y) of a particular multiplier cell (mult) are driven only from a precharge cell (prech1 or prech2), through exactly one transparent latch, with an overall inversion in the path. However, if a logic-0 net (named GND) is encountered in the search, then stop the verification along that path without reporting any violations.

  %path_check -prop { excl_dest inv thru_pipeline_stages exact 1 }
  -except -net GND
  -src -cell prech1
  -src -cell prech2
  -dest -cell mult -pin X
  -dest -cell mult -pin Y
  -seq_xover_via -latch

  The rationale behind constructing complicated rules of this type is that the mult cell may have been optimized for the evaluate transition of the precharge cells, so that there may be a need to prevent usage of the mult cell in other situations. Also, it is possible that the mult cell may have been characterized, for timing, only with the fast evaluate edge in order to reduce pessimism in timing analysis.

- Verify that output pins (DOUT, BDOUT) of certain input pad cells (ioin1, ioin2) are registered only at two types of enabled flip-flops (ff_en1, ff_en2), possibly through some intermediate logic, but do not go anywhere else in the design:

  %path_check -prop { excl_src } *-src -cell ioin1 -pin DOUT*
  -src -cell ioin2 -pin BDOUT
  -dest -cell ff_en1
  -dest -cell ff_en2

- Verify that the signal driving the input (DIN) of a pad output cell (pad_out) is also latched in an observability flip-flop, given that the pad_out cell may not be part of the scan chain for testing:

  %path_check -prop { valid } *-src -net (-cell pad_out -pin DIN) -dest -ff*

- Verify that unregistered output FDOUT of cell ioin1 goes only to an output pad cell (ioout1), possibly through some intermediate logic. This is a case where an external input to a chip is processed and an external output response is generated under some severe time restrictions imposed by the external bus

protocol. So, the unregistered output of cell ioin1 can not be connected anywhere else in the design, except to an output pad cell.

```
%path_check    -prop { excl_src }    -src -cell ioin1 -pin FDOUT
                                     -dest -cell ioout1 -num 1
```

- Verify that an external reset signal, from a primary input net named reset_in, propagates through exactly two pipelined flip-flops before reaching a resettable flip-flop instance /top/xms/rp/dffclr:

```
%path_check    -prop { valid  thru_pipeline_stages exact 2 }
                              -src -net reset_in
                              -dest -inst /top/xms/rp/dffclr
                              -seq_xover_via -ff
```

- Verify the number of elements in a scan chain, between a primary input scan_in and a primary output scan_out, is no more than 1000:

```
%path_check    -prop { valid  thru_pipeline_stages max 1000 }
                              -src -net scan_in
                              -dest -net scan_out
                              -seq_xover_via -ff -pin si
                              -seq_xover_via -ff -pin so
                              -seq_xover_via -ff -pin q
```

Note that the -seq_xover_via option has been used to specify exact pin names for scan input and scan output for all scannable flip-flops in the library, assuming in this case that the scan input pin is named *si* in all flip-flop cells, while the scan output pin is either a dedicated pin named *so* or the normal output named *q*.

A-2.9 Examples of Global Clock Skew Constraint Verification

- Define two clock groups, named “g1” and “g2”, at two specific points within a global clock distribution network:

```
%clk_group g1 /x/y/clk1
%clk_group g2 /x/z/gclk
```

All storage element clock pins driven from /x/y/clk1 become part of group “g1”, and all clock pins driven from /x/z/gclk become part of “g2”.

- Specify an intra-group skew constraint of 50 ps within the clock group “g1”:

%clk_skew_check g1 g1 50

- Specify an inter-group skew constraint of 100 ps between clock groups “g1” and “g2”:

  %clk_skew_check g1 g2 100

APPENDIX: A-3. SAMPLE COMMAND FILES

All of the capabilities offered by SDA can be used in a single run, for maximum effect. The following sample command files illustrate a few common operations that can be performed using SDA. Even though distinct command file examples are used to illustrate various features of the tool, all of these features can be used in any combination in a single run of SDA.

“signal_coupling.cmd”:

```
/* Define timing for clocks, primary inputs, primary outputs */

%wvfm w1 -R 0.0 0.0 1000.0 1000.0 2000.0
%pi_arr a1 0 0 0 0 w1 -L
%pi_arr b1 0 0 0 0 w1 -L
%pi_arr c 0 0 0 0 w1 -L
%pi_arr d 0 20 2000 2020 w1 -L
%pi_clk c w1
%po_req * 800 200 w1 -T

/* Timing analysis controls: delay calculation and signal coupling parameters */

%set_internal_delay_cal -awe_mesh -ceff
%set_coupling_capacitance_multipliers 0.0 1.0 2.0
%set_coupling_window_margins 0 0
%set_small_coupling_capacitance_parameters 0.1 1.0
%set_signal_coupling_analysis

/* Reporting */

%rpt_mode -crit
%rpt_max_slack 5000
%rpt_num_paths 20
%rpt_path_node_details
%rpt_signal_coupling_details
%rpt_mode -hist -slack 100
%rpt_ckt_stats

/* SDF and DSPF outputs */

%sdf_write
```



```
%dspf_write -max

/* Miscellaneous */

%set_multiple_output_files
```

"timing_and_static_constraint.cmd":

```
/*******************************************************************************************/
/* Define clock waveform (period 10 ns) and set 'sclk' as the primary clocks.
   Also, define primary input arrival times.                                               */
/*******************************************************************************************/

%wvfm w1 -R 0 0 5000 5000 10000
%wvfm w2 -R 0 0 5000 5000 10000
%wvfm w3 -R 0 0 5000 5000 10000
%wvfm w4 -R 0 0 15000 15000 30000
%wvfm w5 -R 0 0 15000 15000 30000
%clk /sdram_in_clk w2
%clk /sdram_out_clk w3
%clk /vclks w4
%clk /aclks w5
%pi_clk sclk w1
%pi_arr * 5000 5000 5000 5000 w1 -L
%po_req * 0 0 w1 -L
%set_internal_delay_cal -elmore
%set_wire_load_estimation_mode -lumped_pi_rc -wire_load_model ks_spu

/*******************************************************************************************/
/* Set "setup-time" (min delay) analysis mode and specify violating slack as +500. */
/* Setup for module-to-module critical path                                                */
/*******************************************************************************************/

%set_logic_value -H /gresetN  /* static signal blocking */
%set_logic_value -H /dramcore_mod/if_mod_reset_
%set_logic_value -H /is_dvd3

%rpt_long_names
%rpt_mode -crit
%rpt_max_slack 10000
%rpt_num_paths 20
%rpt_short_path_max_level 6
%rpt_src_module_inst_list { /lcfifo /ldma }
%rpt_dest_module_inst_list { /lcfifo /ldma }

%dspf_write -max
%sdf_write

/*******************************************************************************************/
/* Set local clock group (domains) intra-group  skew.    for top level clk skew       */
/*******************************************************************************************/
/*
%clk_group g1 -local /sclk
```

```
%clk_skew_check g1 g1 200
*/

/*****************************************************************************/
/* Verify wired-or nets are not driven by dis-similar drivers                */
/* tri-state driver EN signal check                                          */
/*****************************************************************************/
/* NOTE: can not directly check wire-or logic from diff. source  */

%func_check -prop { valid similar_drivers } -net_type multi_driver
%func_check -prop { valid mut_excl_drivers } -net_type tristate

/*****************************************************************************/
/* Verify no back-to-back flip-flops.                                        */
/*****************************************************************************/
/* for b2b check for some num. of gates(2 for examp.), use static timing to check */

%path_check -prop {invalid ncombl}
                                        -src -ff -pin q*
                                        -src -latch -pin q*
                                        -dest -ff -pin d*
                                        -dest -ff -pin nen
                                        -dest -latch -pin d*

/*****************************************************************************/
/* Verify flip-flop set/reset pins driven only by global set/reset signals.  */
/*****************************************************************************/

%path_check -prop {excl_dest}           -src -net /gresetN
                                        -dest -ff -pin rb
                                        -dest -latch -pin rb

/* %path_check -prop {excl_dest}        -src -net
                                        -dest -ff -pin sb */

/*****************************************************************************/
/* Maximum fanout(cap or transition) rule for each cell type.                */
/*****************************************************************************/
/* max_fanout is less accurate, max_cap is more appropriate */

%cell_check -cell ckd4   -prop {max_fanout 20} -output */

%cell_check -cell *d1 -except bld1 -prop {max_capacitance 0.15pf} -output
%cell_check -cell *d2 -prop {max_capacitance 0.3pf} -output
%cell_check -cell *d4 -prop {max_capacitance 0.6pf} -output
%cell_check -cell *d8 -prop {max_capacitance 1.2pf} -output
%cell_check -cell *d16 -prop {max_capacitance 2.4pf} -output

%cell_check -cell * -prop {max_transition 1.5ns} -output /* less meaningful if used in pre-
layout */

/*****************************************************************************/
/* Verify no asynchronous logic -- all flip-flops must get clock from primary
   input clock net                                                           */
/*****************************************************************************/
```

```
%path_check -prop {excl_dest}   -src -net sclk
                                -except -src -net /v_fifo/vdd_
                                -except -src -net /a_fifo/vdd_
                                -except -src -net /a_cstrobe
                                -except -src -net /a_fifo/gater_gclk2/q
                                -dest -ff -pin clk

/* -except -src -net /dramcore_mod/if_mod/reset_mod/clkbuf/VDD */

/*****************************************************************************/
/* Verify no floating inputs */
/*****************************************************************************/

%cell_check -all -prop { no_flt_input }


/*****************************************************************************/
/* Misc */
/*****************************************************************************/

%library_check
%rpt_ckt_stats
%set_multiple_output_files
```

"timing_and_static_constraint_alt.cmd":

```
/*****************************************************************************/
/* Define clock waveform (period 10 ns) and set 'sclk' as the primary clocks.
   Also, define primary input arrival times.                                 */
/*****************************************************************************/

%wvfm w1 -R 0 0 5000 5000 10000
%wvfm w2 -R 0 0 5000 5000 10000
%wvfm w3 -R 0 0 5000 5000 10000
%wvfm w4 -R 0 0 15000 15000 30000
%wvfm w5 -R 0 0 15000 15000 30000
%clk /sdram_in_clk w2
%clk /sdram_out_clk w3
%clk /vclks w4
%clk /aclks w5
%pi_clk sclk w1
%pi_arr * 5000 5000 5000 5000 w1 -L
%po_req * 0 0 w1 -L

/* delay calculation */

%set_internal_delay_cal -elmore

/* define arrival times at internal nodes */

%set_arr_time /a1n 100 100 200 200 w1 -L -trans 50 50 75 75
```



```
%set_arr_time /nd2x3/b 1 21 2001 2021 w1 -L

/*****************************************************************************/
/* Set "setup-time" (min delay) analysis mode and specify violating slack as +500. */
/* Setup for module-to-module critical path                                  */
/*****************************************************************************/

%set_logic_value -H /gresetN  /* static signal blocking */
%set_logic_value -H /dramcore_mod/if_mod_reset_
%set_logic_value -H /is_dvd3

%rpt_long_names
%rpt_mode -crit
%rpt_max_slack 10000
%rpt_num_paths 20
%rpt_short_path_max_level 6
%rpt_src_module_inst_list { /lcfifo /ldma }
%rpt_dest_module_inst_list { /lcfifo /ldma }

/*****************************************************************************/
/* Set local clock group (domains) intra-group  skew.    for top level clk skew      */
/*****************************************************************************/
/*
%clk_group g1 -local /sclk
%clk_skew_check g1 g1 200
*/

/*****************************************************************************/
/* Verify wired-or nets are not driven by dis-similar drivers                */
/* tri-state driver EN signal check                                          */
/*****************************************************************************/
/* NOTE: can not directly check wire-or logic from diff. source  */

%func_check -prop { valid similar_drivers } -net_type multi_driver
%func_check -prop { valid mut_excl_drivers } -net_type tristate

%func_check -prop { valid similar_drivers } -net /lmac/lmac_TDO29
%func_check -prop { valid similar_drivers } -net /lzmem/lzmaddr_t_gbus_8_
%func_check -prop { valid similar_drivers } -net /lpmem/lpmaddr_t_gbus_9_

/*****************************************************************************/
/* Verify no back-to-back flip-flops.                                        */
/*****************************************************************************/
/* for b2b check for some num. of gates(2 for examp.), use static timing to check */

%path_check -prop {invalid ncombl}
                                        -src -ff -pin q*
                                        -src -latch -pin q*
                                        -dest -ff -pin d*
                                        -dest -ff -pin nen
                                        -dest -latch -pin d*

/*****************************************************************************/
/* Verify flip-flop set/reset pins driven only by global set/reset signals.    */
/*****************************************************************************/
```

```
%path_check -prop {excl_dest}                    -src -net /gresetN
                                          -dest -ff -pin rb
                                          -dest -latch -pin rb

/*****************************************************************************/
/* Maximum fanout(cap or transition) rule for each cell type.                */
/*****************************************************************************/

/* max_fanout is less accurate, max_cap is more appropriate */

%cell_check -cell ckd4   -prop {max_fanout 20} -output */

%cell_check -cell *d1 -except bld1 -prop {max_capacitance 0.15pf} -output
%cell_check -cell *d2 -prop {max_capacitance 0.3pf} -output
%cell_check -cell *d4 -prop {max_capacitance 0.6pf} -output
%cell_check -cell *d8 -prop {max_capacitance 1.2pf} -output
%cell_check -cell *d16 -prop {max_capacitance 2.4pf} -output

%cell_check -cell * -prop {max_transition 1.5ns} -output /* less meaningful if used in pre-
layout */

/*****************************************************************************/
/* Verify no asynchronous logic -- all flip-flops must get clock from primary
  input clock net                                                          */
/*****************************************************************************/

%path_check -prop {excl_dest}   -src -net sclk
                                -except -src -net /v_fifo/vdd_
                                -except -src -net /a_fifo/vdd_
                                -except -src -net /a_cstrobe
                                -except -src -net /a_fifo/gater_gclk2/q
                                -dest -ff -pin clk
                                -except -src -net /dramcore_mod/if_mod/reset_mod/clkbuf/VDD

/*****************************************************************************/
/* Verify no floating inputs */
/*****************************************************************************/

%cell_check -all -prop { no_flt_input }


/*****************************************************************************/
/* Automatic clock gate setup/hold check */
/*****************************************************************************/

%clk_gate_data_setup -cell_type cell_and -R -R 50
%clk_gate_data_setup -cell_type cell_and -F -R 500
%clk_gate_data_hold -cell_type cell_and -F -F 50


/*****************************************************************************/
/* Data-to-clockwvfm and data-to-data setup/hold checks */
/*****************************************************************************/
```

```
%setup_check /ix2/a -B w1 -T 800
%hold_check /ix2/a -B w1 -T 200

%setup_check /pqr/x30/B -F /pqr/x30/A -R 2000
%hold_check /pqr/x30/B -R /pqr/x30/A -F 1000
```

"path_spice_deck_2paths.cmd":

```
/* define clock waveform and primary input/output timing characteristics */

%wvfm w1 -R 0.0 0.0 1000.0 1000.0 2000.0
%pi_arr a1 0 0 0 0 w1 -L
%pi_arr b1 0 0 0 0 w1 -L
%pi_arr c 0 0 0 0 w1 -L
%pi_arr d 0 20 2000 2020 w1 -L
%pi_clk c w1
%po_req * 800 200 w1 -T

/* select internal delay calculation mode */

%set_internal_delay_cal -awe -ceff


/* timing report options */

%rpt_path_node_details
%rpt_mode -crit
%rpt_max_slack 5000
%rpt_num_paths 20
%rpt_path y1
%rpt_path n3 z1

/* spice decks for top 2 user-defined paths and top 2 critical paths */

%set_path_spice_deck_gen -rpt_paths 2
%set_path_spice_deck_gen -crit_paths 2


/* misc. */

%set_multiple_output_files
%rpt_ckt_stats
```

"path_spice_deck_all.cmd":

```
/* define clock waveform and primary input/output timing characteristics */

%wvfm w1 -R 0.0 0.0 1000.0 1000.0 2000.0
%pi_arr a1 0 0 0 0 w1 -L
%pi_arr b1 0 0 0 0 w1 -L
%pi_arr c 0 0 0 0 w1 -L
```

```
%pi_arr d 0 20 2000 2020 w1 -L
%pi_clk c w1
%po_req * 800 200 w1 -T


/* select internal delay calculation mode */

%set_internal_delay_cal -awe -ceff


/* timing report options */

%rpt_path_node_details
%rpt_mode -crit
%rpt_max_slack 5000
%rpt_num_paths 20
%rpt_path y1
%rpt_path n3 z1


/* spice decks for all user-defined paths and all critical paths */

%set_path_spice_deck_gen -rpt_paths -all
%set_path_spice_deck_gen -crit_paths -all


/* misc. */

%set_multiple_output_files
%rpt_ckt_stats
```

"logic_cone_spice_deck.cmd":

```
/* generate logic cone spice deck starting from node 'clk' */

%set_logic_cone_spice_deck_gen clk


/* use multiple output files */

%set_multiple_output_files


/* define spice subcircuit pin orders for various cell types */

%spice_subckt dc_14 O0 I1 I0
%spice_subckt dc_16 O0 I1
%spice_subckt dc_18 O0 I0
%spice_subckt dc_20 O0 I0
%spice_subckt dc_21 O0 I0
%spice_subckt dc_15 O0 I0
%spice_subckt dc_17 O0 I0
%spice_subckt dc_19 O0 I0
```



```
%spice_subckt dc_4 O0 I0
%spice_subckt dc_8 O0 I0
```

"timing_with_signal_coupling.cmd":

```
/* define clock waveform and primary input/output timing characteristics */

%wvfm w1 -R 0 0 5000 5000 10000
%pi_clk sclk w1
%pi_arr * 5000 5000 5000 5000 w1 -L
%po_req * 0 0 w1 -L

/* select internal delay calculation mode */

%set_internal_delay_cal -awe

/* duplicate coupling capacitors (if dspf file includes coupling on only one
   of the two nets) */

%duplicate_spf_cc


/* signal coupling analysis */

%set_coupling_capacitance_multipliers 0.0 1.0 2.0
%set_coupling_window_margins 0 0
%set_signal_coupling_analysis


/* timing reports */

%rpt_mode -crit
%rpt_max_slack 5000
%rpt_num_paths 20


/* critical path spice decks */

%set_path_spice_deck_gen -crit_paths 4


/* misc. */

%set_multiple_output_files
%rpt_ckt_stats
```

"timing_with_signal_coupling_sdfout.cmd":

```
/* define clock waveform and primary input/output timing characteristics */
```

```
%wvfm w1 -R 0 0 5000 5000 10000
%pi_clk sclk w1
%pi_arr * 5000 5000 5000 5000 w1 -L
%po_req * 0 0 w1 -L


/* select internal delay calculation mode */

%set_internal_delay_cal -awe


/* signal coupling analysis (coupling caps not duplicated) */

%set_coupling_capacitance_multipliers 0.0 1.0 2.0
%set_coupling_window_margins 0 0
%set_signal_coupling_analysis


/* SDF output */
%sdf_write
```

"buffer_eco_setup.cmd":

```
/* define clock waveform and primary input/output timing characteristics */

%wvfm w1 -R 0.0 0.0 1000.0 1000.0 2000.0
%pi_arr a1 0 0 0 0 w1 -L
%pi_arr b1 0 0 0 0 w1 -L
%pi_arr c 0 0 0 0 w1 -L
%pi_arr d 0 20 2000 2020 w1 -L
%pi_clk c w1
%po_req * 800 200 w1 -T

/* select internal delay calculation mode */

%set_internal_delay_cal -awe_mesh -ceff


/* select ECO parameters for setup-time optimization */

%set_sizing_parameters 1.5 1.5 1.5 6.0
%set_buffering_parameters 0.5 3.0 3.0 2.0 3.0 500
%use_buffer_cell_type buf_4
%optimize_gate_sizes -max_cap -min_slack 0 -enable_buffer_insertion


/* timing reports: top 20 critical paths (slack <= 5000 ps), point-to-point user-
                defined paths and histogram. */

%rpt_mode -crit
%rpt_max_slack 5000
%rpt_num_paths 20
```



```
%rpt_mode -hist -slack 100

/* SDF output */

%sdf_write

/* use multiple output files */

%set_multiple_output_files
```

"buffer_eco_hold.cmd":

```
/* define clock waveform and primary input/output timing characteristics */

%wvfm w1 -R 0.0 0.0 1000.0 1000.0 2000.0
%pi_arr a1 -1000 -1000 -1000 -1000 w1 -L
%pi_arr b1 0 0 0 0 w1 -L
%pi_arr c 0 0 0 0 w1 -L
%pi_arr d 0 20 2000 2020 w1 -L
%pi_clk c w1
%po_req * 800 200 w1 -T

/* select internal delay calculation mode */

%set_internal_delay_cal -awe_mesh -ceff

/* select ECO parameters for hold-time optimization */

%set_sizing_parameters 1.5 1.5 1.5 6.0
%set_buffering_parameters 1.5 1.5 1.5 4.0 1.5 500
%use_buffer_cell_type buf_4
%optimize_gate_sizes -fix_hold_time -min_hold_slack 2000 -min_setup_slack 200

/* timing reports: top 20 critical paths (slack <= 5000 ps), point-to-point user-
                   defined paths and histogram. */

%rpt_mode -crit
%rpt_max_slack 5000
%rpt_num_paths 20
%rpt_path /n2 /n3 /n4 /z1
%rpt_path /c /n3 /n4 /z1
%rpt_mode -hist -slack 100

/* SDF output */

%sdf_write
```



```
/* use multiple output files */

%set_multiple_output_files
```

APPENDIX: A-4. SDA INSTALLATION AND LICENSE SETUP

The SDA software package is available for downloading on our web site at www.suryatech.com.

The latest binary executables (for the 64-bit Sparc Solaris and the 32-bit Pentium Linux platforms) and the user guide can be downloaded from: http://www.suryatech.com/downloads/releases.

Note that the binary files are gzipped. You will also need a valid FLEXlm license file in order to run SDA. The user guides are in PDF format. You will need a password to open the PDF files.

The FLEXlm license daemon for the 64-bit Sparc Solaris and the 32-bit Pentium Linux platforms can be downloaded from http://www.suryatech.com/downloads/Surya_License_Daemons.

For additional assistance with installation or usage of SDA, please contact customer_service@suryatech.com.

FLEXlm v8.0d or later version is required for license management. For additional information on floating license management based on FLEXlm, please see www.globetrotter.com.

