Standard Delay Format Specification

Version 3.0

May 1995

Open Verilog International

Contents

Introduction

Introduction

The Standard Delay Format (SDF) file stores the timing data generated by EDA tools for use at any stage in the design process. The data in the SDF file is represented in a tool-independent way and can include

- Delays: module path, device, interconnect, and port
- Timing checks: setup, hold, recovery, removal, skew, width, period, and nochange
- Timing constraints: path, skew, period, sum, and diff
- Timing environment: intended operating timing environment
- Incremental and absolute delays
- Conditional and unconditional module path delays and timing checks
- Design/instance-specific or type/library-specific data
- Scaling, environmental, and technology parameters

Throughout a design process, you can use several different SDF files. Some of these files can contain pre-layout timing data. Others can contain path constraint or post-layout timing data.

The name of each SDF file is determined by the EDA tool. There are no conventions for naming SDF files.

Introduction to Version 3.0

Version 3.0 of the Standard Delay Format includes many enhancements for the specification of the environment in which a circuit is operating with regard to timing. Along with existing and new constraint information, this makes the format much more useful for communication between timing analysis and synthesis tools.

Some new constructs and enhancements for the back-annotation of computed timing data are also included. For example, the “removal” timing check bears the same relationship to a recovery check as the hold check does to a setup check. Note that some of the new constructs anticipate corresponding enhancements to popular analysis tools.

Many SDF files written to the 2.1 specification will also conform to the 3.0 version (with the adjustment of the **SDFVERSION** entry). However, some significant changes in the area of constraints and the less frequently used back-annotation constructs mean that the new format is not 100% backward-compatible. File readers should use the **SDFVERSION** entry if they are unable to adapt to the differences automatically. See the version history later in this chapter for complete information about changes.

Published by OVI

OVI has developed this SDF specification to enable accurate and unambiguous transfer of delay data between tools that require timing. ***All parties utilizing the SDF should interpret and manipulate delay data according to this specification.*** The specification will be provided free of charge to all interested members of OVI. ASIC Vendors and 3rd party tool suppliers that desire copies of the SDF specification should request it from the OVI headquarters. Please direct your requests to:

Lynn Horobin
Open Verilog International
15466 Los Gatos Blvd., Suite 109-071,
Los Gatos, CA 95032

Tel: (408) 353-8899
Fax: (408) 353-8869
internet e-mail: ovi@netcom.com

Acknowledgements

The OVI Logic Modeling Technical SubCommittee acknowledges the individual and team efforts invested in establishing this version of the SDF specification:

Brien Anderson (Acuson)

Tim Ayres (Zycad)

Bruce Bandali (LSI Logic)

John Busco (Toshiba)

Irene Chang (Mitsubishi)

Shir-Shen Chang (Synopsys)

Graham Davies (Viewlogic)

Ted Elkind (Cadence)

Vassilios Gerousis (Motorola)

Hector Lai (Quad Design)

Jimmy Lin (Toshiba)

Liren Liu (Actel)

Ying-li Ren (Lattice)

Steven Sliman (Texas Instruments)

Hoanh Tran (Toshiba)

Version History

Version 2.0 - June, 1993

- The keywords, **USERDEF** and **INCLUDE**, which were in version 1.0, are no longer supported by OVI SDF 2.0.
- Hierarchy divider character restricted to period (.) or slash (/).
- Use of **COND** keyword with timing checks revised and *timing_check_condition* restricted for correspondence with the Verilog language.
- **CORRELATION** construct added to **CELL**.
- C and C++ style comments now allowed in SDF files.
- Alterations to all examples of the **RECOVERY** timing check, unfortunately resulting in them being **incorrect** in version 2.0 of the specification, see version 2.1, below.
- **WIDTH** and **PERIOD** construct descriptions corrected - width and period timing checks are for **minimum** allowable pulse width and period, not **maximum**.
- All delay constructs (**IOPATH**, **DEVICE**, **PORT**, **INTERCONNECT** and **NETDELAY**) changed to permit negative values instead of only positive, *value* changed to *rvalue* in formal syntax descriptions involving these keywords.
- Corrections to **TIMINGCHECK** entries in Example 2 of Section 2.
- Other minor changes to descriptive text.

Version 2.1 - February, 1994

- Formal syntax description consolidated in new chapter, BNF symbols *value* and *exp_list* deleted, *absvals* and *incvals* both replaced by *del_def*, some symbols changed for more intuitive reading, other minor corrections and reorganizations.
- The **SDFVERSION** entry in the header is now required. Other entries in the header are still optional, but, if present, must now contain data (i.e. “empty” entries such as **(DESIGN)** are no longer allowed).
- The use of the wildcard instance specification restricted to cells at the ASIC physical primitive level, no longer allowed in PATH or *port_path*.
- Description of **CORRELATION** entry expanded and syntax revised to avoid possible confusion with “min:typ:max” triples.
- **CELL** entries may now have zero or more *timing_spec*s (previously one or more), allowing **CELL** entries to carry a **CORRELATION** entry without other timing data.
- The option to provide a single value or a “min:typ:max” triple has been made available uniformly to all constructs. However, it is now prohibited to mix single values and triples in the same SDF file.
- The semantics of delay values (*rvalue*s) in an *rvalue_list* (formerly *rvalue*) has been defined for lists of length 1, 2, 3, 6 or 12. Provision is made for omitting *rvalue*s from the ends of lists of 6 and 12. Any *rvalue* can be null.

- **PATHPULSE** and **GLOBALPATHPULSE** changed to allow specification of both ports of a path or neither, the latter applying to all paths across the cell.
- Improved description of use of *port_instance* specification in **DEVICE** entries to apply delays to paths ending at a particular output port.
- *timing_check_condition* now allows only ~ and ! operators to be applied to *scalar_port*s rather than the full set of UNARY_OPERATORs.
- **WIDTH** and **PERIOD** entries restricted to have only non-negative values, formal syntax definition changed to reflect this.
- Improved description of **RECOVERY** timing check and correction of all examples (asynchronous control signal port reference comes before clock port reference).
- Description of **DIFF** entry corrected to agree with formal syntax description; only two paths are permitted, not more. **SUM** and **DIFF** now allow two data values to differentiate rise and fall delays. Since only positive values make sense for the **DIFF** constraint, this is now enforced by the syntax.
- Improved description of **SKEWCONSTRAINT** construct. Since only positive values make sense for this constraint, this is now enforced by the syntax.
- Proposal for SDF version 3.0 revised.
- Other extensions and changes to descriptive text.

Correction to Version 2.1 - July, 1994

- The formal syntax for *rvalue_list* was incorrect as originally published in that it implied a second set of parentheses around each *rvalue* in the list. The correction consists of a revision to page 4-7 removing these extra parentheses and improving the explanatory text. All examples of the usage of *rvalue_list* were correct as originally published.

Version 3.0 - May, 1995

- The alternative for identifying cell instances by repetition of the *instance* construct removed. The more compact method using a single **INSTANCE** keyword followed by a PATH (using the hierarchy divider character) is now required in all cases where a specific region of the design is to be identified.
- The two alternatives for identifying specific ports of specific cell instances rationalized into one method. The syntax of the old *port_instance* symbol, using the **INSTANCE** keyword and a PATH in parentheses before the port name, has been eliminated. The syntax of *port_path*, using a PATH, the hierarchy delimiter and the port name, has been retained. However, to reduce confusion between paths though the design hierarchy and circuit paths that have timing data associated with them, the BNF symbol *port_instance* has been adopted for this syntax.
- The restriction of the use of the wildcard instance specification to cells at the ASIC physical primitive level added in SDF 2.1 was removed.
- **CORRELATION** construct removed.
- **GLOBALPATHPULSE** keyword changed to **PATHPULSEPERCENT**.
- Descriptions of **PATHPULSE** and **PATHPULSEPERCENT** much expanded to include an explanation of the intended analysis tool operation.

- Pulse propagation limits may now be specified in all delay constructs using *delval* and *delval_list*, which extend the *rvalue_list* of previous versions.
- An optional symbolic name can now appear after the **COND** keyword (and the new **SCOND** and **CCOND** keywords) to stand in for the state or condition expression to assist annotators that operate by matching named placeholders.
- **CONDELSE** keyword added to allow specification of default delays over state-dependent input-output paths.
- **NETDELAY** construct removed.
- **RETAIN** added to represent the time for which an output/bidirectional port will retain its previous logic value after a change at a related output/ bidirectional port.
- Negative values in timing check constructs **SETUP**, **HOLD** and **RECOVERY** disallowed.
- Alternative syntax for **SETUPHOLD** added.
- **REMOVAL** added. This new construct is to **RECOVERY** what **HOLD** is to **SETUP**.
- **RECREM** added. This is the combination of **RECOVERY** and **REMOVAL** in the same way as **SETUPHOLD** combines **SETUP** and **HOLD**. This new construct allows syntax similar to both versions of **SETUPHOLD** and permits negative limit values for the recovery or removal time subject to the constraint that their sum be greater than zero.
- A new timing specification construct, Timing Environment, with the keyword **TIMINGENV**, added to SDF at the same level as **DELAY** and **TIMINGCHECK** sections. The constraint constructs, **PATHCONSTRAINT**, **SUM**, **DIFF** and **SKEWCONSTRAINT**, moved to this section from the Timing Check section. Several entirely new constructs added here.
- Optional *name* added to **PATHCONSTRAINT** construct to allow a symbolic name to be associated with a path.
- **PERIODCONSTRAINT** environment construct added to specify a path constraint value for groups of paths in a synchronous circuit.
- **WAVEFORM** environment construct added to describe input waveforms.
- **ARRIVAL** environment construct added to specify the time at which a primary input signal is to be applied during the intended circuit operation.
- **DEPARTURE** environment construct added to specify the time at which a primary output signal is to occur during the intended circuit operation.
- **SLACK** environment construct added to specify available margin in a delay path.

2

SDF in the Design Process

SDF in the Design Process

Sharing of Timing Data

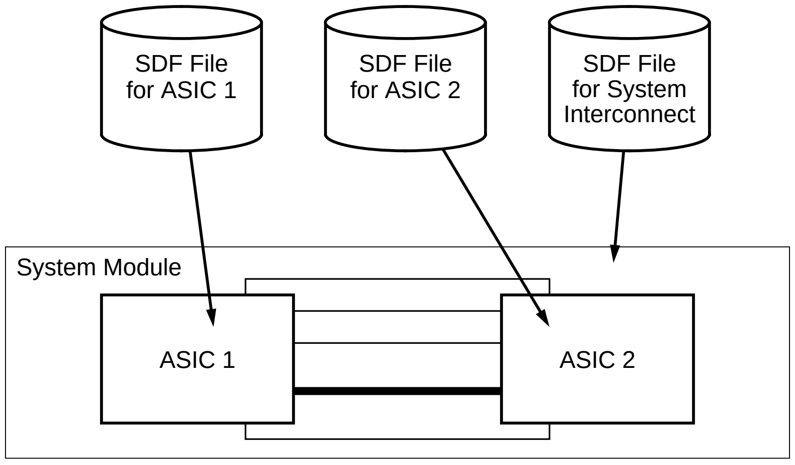
By accessing an SDF file, EDA tools are assured of consistent, accurate, and up-to-date data. This means that EDA tools can use data created by other tools as input to their own processes. Sharing data in this way, layout tools can use design constraints identified during timing analysis, and simulation tools can use the postlayout delay data.

The EDA tools create, read (to update their design), and write to SDF files.

Using Multiple SDF Files in One Design

SDF files support hierarchical timing annotation. A design hierarchy might include several different ASICs (and/or cells or blocks within ASICs), each with its own SDF file, see Figure 1.

Figure 1 Multiple SDF Files in a Hierarchical Design



Timing Data and Constraints

SDF contains constructs for the description of computed timing data for back-annotation and the specification of timing constraints for forward-annotation. There is no restriction on using both sets of constructs in the same file. Indeed, some design synthesis tools (such as floorplanning) may need access to computed timing data as well as the timing constraints they are intended to meet.

The following sections discuss the use of SDF for back- and forward-annotation of timing information.

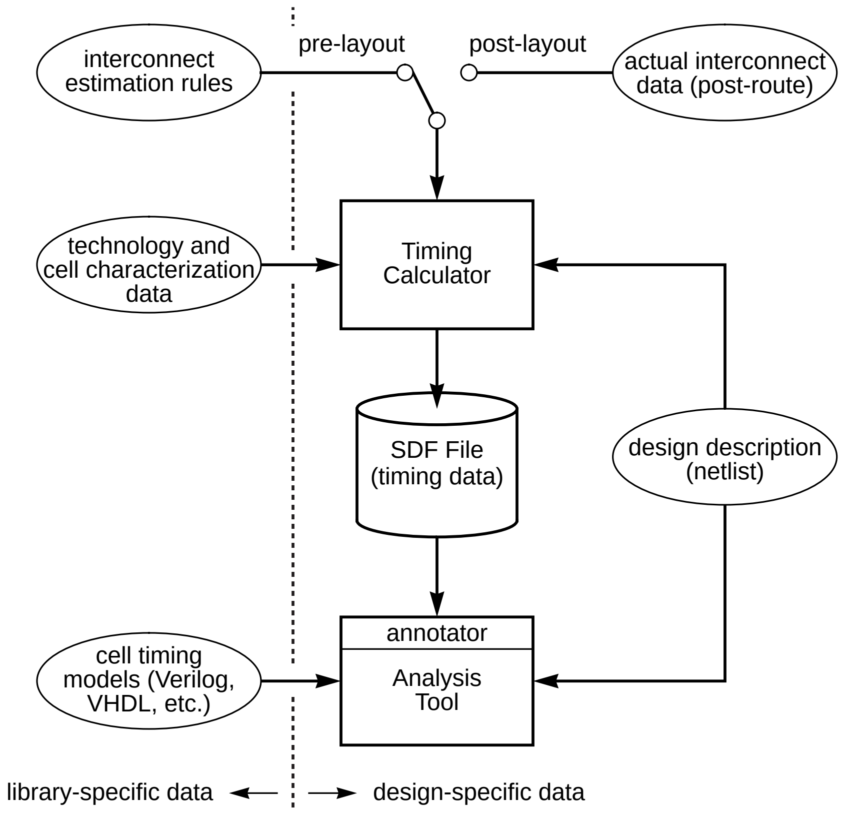
Timing Environment

SDF includes constructs for describing the intended timing environment in which a design will operate. For example, you can specify the waveform to be applied at clock inputs and the arrival time of primary inputs.

Back-Annotation of Timing Data for Design Analysis

Figure 2 shows the use of SDF in back-annotating timing data to an analysis tool. An advantage of this approach is that once an SDF file has been created for a design, all analysis and verification tools can access the same timing data, which ensures consistency. Note, however, that different tools may have different restrictions in the way in which they use the data in an SDF file. For example, static timing analysis tools may be able to take into account path delays which have a negative value, whereas dynamic timing simulation tools may have to interpret such negative delays as zero. Thus, although by using SDF the timing data used by each tool is the same, differences in tool capabilities may nevertheless result in small differences in analysis results.

Figure 2 SDF Files in Timing Back-Annotation


interconnect estimation rules
pre-layout
post-layout
actual interconnect data (post-route)
technology and cell characterization data
Timing Calculator
SDF File (timing data)
design description (netlist)
cell timing models (Verilog, VHDL, etc.)
annotator
Analysis Tool
library-specific data
design-specific data

The Timing Calculator

A timing calculator tool is responsible for generating the SDF file. To do this, it will examine the specific design for which it has been instructed to calculate timing data. In the figure, this is illustrated by the arrow from the design description (netlist). The timing calculator must locate, within the design, each region for which a timing model exists and calculate values for the parameters of that timing model. Strategies for doing this vary from technology to technology, but an example would be the location of each occurrence of a physical primitive from an ASIC library and the calculation of its timing properties at its boundary (pin-to-pin timing). Knowledge of the timing models may be obtained by accessing them directly (not shown) or may be built into the timing calculator and/or cell characterization data.

As the timing characteristics of ASICs are strongly influenced by interconnect effects, the figure shows the timing calculator using estimation rules (pre-layout) or actual interconnect data (post-layout). Thus, SDF is suitable for both pre-layout and post-layout application.

The timing data for the design is written by the timing calculator into the SDF file. SDF imposes no restrictions on the precision to which the data is represented. Therefore, the accuracy of the data in the SDF file will be dependent on the accuracy of the timing calculator and the information made available to it, such as pre-layout interconnect estimation methods or post-layout interconnect data extracted from the device topology.

The Annotator

The SDF file is brought into the analysis tool through an annotator. The job of the annotator is to match data in the SDF file with the design description and the timing models. Each region in the design identified in the SDF file must be located and its timing model found. Data in the SDF file for this region must be applied to the appropriate parameters of the timing model.

The annotator may be instructed to apply the data in the SDF file to a specific region of the design, other than at the top level of the design hierarchy. In this case, it will search for regions identified in the SDF file starting at this point in the hierarchy. The file must clearly have been prepared with this in mind, otherwise the annotator will be unable to match what it finds in the file with the design viewed from this point.

The foregoing implies that the annotator must have access to the design description and the timing models. Frequently, this will be via the internal representations maintained by the analysis tool. The annotator will then be a part of the tool. As an alternative, the annotator may operate independently of the analysis tool and convert the data in the SDF file into a format suitable for the tool to read directly. If such an annotator is unable to match the SDF file to the design description and the timing models, then the effect of inconsistencies may be unpredictable. Also, certain constructs of SDF cannot be supported without access to the design description (for example, wildcard cell instance specifications).

Consistency Between SDF File and Design Description

An SDF file contains timing data for a specific design. The contents of the file identifies regions of the design and provides timing data that applies to the timing properties of that region. The analysis tool or annotator cannot operate if the regions identified in the SDF file do not correspond exactly with the design description. Therefore, changes to the design generally require the timing calculator to be re-run and a new SDF file to be written.

Of equal importance to the logic of the design is the naming of design objects. Even if the same cells are present and are connected in the same way, annotation cannot operate if the names by which these cells and nets are known differ in the SDF file and design description. The naming of objects must be consistent in these two places.

During annotation, inconsistencies between the SDF file and the design description are considered errors.

Consistency Between SDF File and Timing Models

An SDF file contains only timing data. It does not contain instructions to the analysis tool as to how to model the timing properties of the design. The SDF keywords and constructs which surround the data in the file describe the timing relationships between elements in the design only so that the data can be identified by the annotator and applied to the timing model in the correct way. It is assumed that the timing models used by the design are described to the analysis tool by some means other than the SDF file. Thus, when using SDF, it is crucial that the data in the SDF file is consistent with the timing models. For example, if the SDF file identifies an occurrence of a 2-input NAND gate ASIC library cell in the design and states that the input-output path delay from the A input to the Y output is 0.34ns, then it is imperative that the timing model for this cell should have an input port A, an output port Y and that the cell’s delays are described in terms of pin-to-pin delays (as opposed, for example, to distributed delays or a single all-inputs-to-the-output delay).

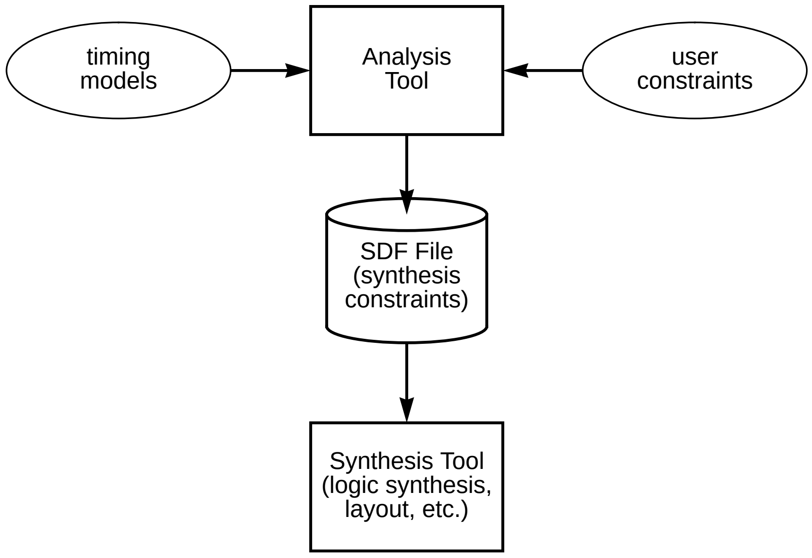
Some analysis tools and their annotators can extend the timing models in certain ways. Specifically, an interconnect timing model is often not explicitly stated in the cell timing models or in the design description. The tool and/or annotator conspire to add this information when the design and timing are loaded or merged in the tool. In this case, the SDF file will contain data that has no obvious “place to go” in the models. Nevertheless, the data must be consistent with the tool’s capabilities to model circuit timing using that data. For example, if you describe interconnect timing in the SDF file in a point-to-point fashion, but the analysis tool can only represent interconnect timing as delay at cell inputs, then the tool may reject this data or perform a mapping to input delays, possibly losing information in the process.

During annotation, inconsistencies between the SDF file and the timing models are considered errors.

Forward-Annotation of Timing Constraints for Design Synthesis

In addition to the back-annotation of timing data for analysis, SDF supports the forward-annotation of timing constraints to design synthesis tools. (Here, we use the term "synthesis" in its broad sense of construction, thus including not only logic synthesis, but floorplanning, layout and routing.) Timing constraints are "requirements" for the design's overall timing properties, often modified and broken down by previous steps in the design process. Figure 3 shows a typical scenario of SDF in a design synthesis environment.

Figure 3 SDF Files in Constraint Forward-Annotation


timing models
Analysis Tool
user constraints
SDF File (synthesis constraints)
Synthesis Tool (logic synthesis, layout, etc.)

For example, the initial requirement might be that the primary clock should run at 50MHz. A static timing analysis of the design might identify the critical paths and the available "slack" time on these paths and pass constraints for these paths to the floorplanning, layout and routing (physical synthesis) tools so that the final design is not degraded beyond the requirement. Alternatively, if after layout and routing, the requirement cannot be met, constraints for the problem paths might be constructed and passed back to a logic synthesis tool so that it can "try again" and leave more slack for physical synthesis.

Constraints may also be originated by an analysis tool alone. Consider a synchronous system in which the clock distribution system is to be synthesized. A static timing analysis may be able to determine the maximum permissible skew over the distribution network and provide this as a constraint to clock synthesis. In turn, this tool may break down the skew constraint into individual path constraints and forward this to physical synthesis.

Note :- the term “timing constraint” is also in use to describe what in SDF are called timing checks. When viewed as statements of the form “this condition must be met or the circuit won’t work”, they are indeed the same. Perhaps the only distinction is that timing checks are applied to an analysis tool, which is only in a position to check to see if they are met and indicate a violation if they are not, whereas constraints are applied to a synthesis tool, which may adapt its operation to ensure that the specified condition is met.

In this specification, we use “timing check” to mean a test that an analysis tool performs to make sure that a circuit, as presently constructed, will operate reliably. We use “timing constraint” or “constraint” to mean a restriction on the timing properties of a design that we specify to a tool that is going to construct or modify some aspect of the design (e.g. logic, layout or routing).

Timing Models Supported by SDF

The importance of the consistency of an SDF file with the timing models has been stressed above. SDF provides a variety of ways in which the timing of a circuit can be described, allowing considerable flexibility in the design of the timing models. This section describes some modeling methodologies supported by SDF and establishes a consistent terminology that we will use later in describing SDF itself.

Modeling Circuit Delays

SDF supports both a pin-to-pin and a distributed delay modeling style.

A pin-to-pin modeling style is generally one in which each physical cell in an ASIC library has its timing properties described at its boundary, i.e. with direct reference only to the ports of the cell. The timing model is frequently distinct from the functional part of the model and has the appearance of a “shell”, intercepting transitions entering and leaving the functional model and applying appropriate delays to output transitions. The SDF **IOPATH** construct is intended to apply delay data to input-to-output path delays across cells described in this way. The **COND** construct allows any path delay to be made conditional, that is, its value applies only when the specified condition is true. This allows for state-dependency of path delays where the path appears more than once in the timing model with conditions to identify the circuit state when it applies.

A distributed modeling style is generally one in which the timing properties of the cell are embedded in the description of the cell as a network of modeling primitives. The primitives provided by analysis tools such as simulators and timing analyzers usually have simple timing capabilities built into them, such as the ability to delay an output signal transition. The delay properties of the cell are constructed by the careful arrangement of modeling primitives and their delays. The SDF **DEVICE** construct is intended to apply delay data to modeling primitives in distributed delay models.

Modeling Output Pulse Propagation

SDF supports the specification of how short pulses propagate to the output of a cell described using a pin-to-pin delay model. A limit can be established for the shortest pulse that will affect the output and a larger limit can be established for the shortest pulse that will appear with its true logical value, rather than appearing as a “glitch” to the unknown state. The SDF **PATHPULSE** construct allows these limits to be specified as time values. The SDF **PATHPULSEPERCENT** construct allows these limits to be specified as percentages of the path delay.

Modeling Timing Checks

SDF supports setup, hold, recovery, removal, maximum skew, minimum pulse width, minimum period and no-change timing checks. Library models can specify timing checks with respect to both external ports and internal signals. Negative values are permitted on timing checks where this is meaningful, although analysis tools that cannot use negative values may substitute a value of zero. The SDF **COND** construct allows conditional timing checks to be specified.

Modeling Interconnect Delays

SDF supports two styles of interconnect delay modeling.

The SDF **INTERCONNECT** construct allows interconnect delays to be specified on a point-to-point basis. This is the most general method of specifying interconnect delay.

The SDF **PORT** construct allows interconnect delays to be specified as equivalent delays occurring at cell input ports. This results in no loss of generality for wires/nets that have only one driver. However, for nets with more than one driver, it will not be possible to represent the exact delay over each driving-output-to-driven-input path using this construct. Note that for timing checks to operate correctly when interconnect is modeled in this way, the timing models must be constructed to apply the delay to the signal at input ports before they arrive at the timing checks.

Using “Internal” Nodes

SDF allows ports to be specified which are neither external connections of an ASIC library physical primitive nor connections between levels in the design hierarchy. Such “internal nodes” may have no corresponding terminal or net in the physical design but may instead be artifacts of the way the timing and/or functional model is constructed. For specific tools, the use of internal nodes can increase the flexibility and accuracy of the models. However, because the annotator must be able to match data in the SDF file to the models, SDF files referencing internal nodes will not be portable to tools that do not share the same concept of internal nodes or have models constructed without or with different internal nodes.

3

Using the Standard Delay Format

SDF File Content

This chapter describes the content of an SDF file. For each part of the file, the purpose is discussed, the syntax is specified and an example is presented. A complete, formal definition of the file syntax is contained in a separate chapter. You may wish to refer to that chapter for precise definitions of some of the abbreviated syntax descriptions given here.

SDF files are ASCII text files. Every SDF file contains a header section followed by one or more cell entries.

Syntax

delay_file ::= **(DELAYFILE** *sdf_header cell+* **)**

The header section, *sdf_header*, contains information relevant to the entire file such as the design name, tool used to generate the SDF file, parameters used to identify the design and operating conditions (see “Header Section” on page 3).

Each cell entry, *cell*, identifies part of the design (a “region” or “scope”) and contains data for delays, timing checks, constraints and the timing environment (see “Cell Entries” on page 8). A *cell* may be a physical primitive from the ASIC library, a modeling primitive for a specific analysis tool or some user-created part of the design hierarchy. In fact, a *cell* may encompass the entire design.

Example

```
(DELAYFILE
  (SDFVERSION  "3.0")
  (DESIGN      "BIGCHIP")
  (DATE        "March 12, 1995 09:46")
  (VENDOR      "Southwestern ASIC")
  (PROGRAM     "Fast program")
  (VERSION     "1.2a")
  (DIVIDER     /)
  (VOLTAGE     5.5:5.0:4.5)
  (PROCESS     "best:nom:worst")
  (TEMPERATURE -40:25:125)
  (TIMESCALE   100 ps)
  (CELL
    (CELLTYPE "BIGCHIP")
    (INSTANCE top)
    (DELAY
      (ABSOLUTE
        (INTERCONNECT mck b/c/clk (.6:.7:.9))
        (INTERCONNECT d[0] b/c/d (.4:.5:.6))
      )
    )
  )
  (CELL
    (CELLTYPE "AND2")
    (INSTANCE top/b/d)
    (DELAY
      (ABSOLUTE
        (IOPATH a y (1.5:2.5:3.4) (2.5:3.6:4.7))
        (IOPATH b y (1.4:2.3:3.2) (2.3:3.4:4.3))
      )
    )
  )
  (CELL
    (CELLTYPE "DFF")
    (INSTANCE top/b/c)
    (DELAY
      (ABSOLUTE
        (IOPATH (posedge clk) q (2:3:4) (5:6:7))
        (PORT clr (2:3:4) (5:6:7))
      )
    )
    (TIMINGCHECK
      (SETUPHOLD d (posedge clk) (3:4:5) (-1:-1:-1))
      (WIDTH clk (4.4:7.5:11.3))
    )
  )
  (CELL
    . . .
  )
)
```

header section

cell 1

cell 2

cell 3

cell 4
.
.
cell n

Header Section

The header section of an SDF file contains information that relates to the file as a whole. Except for the SDF version, entries are optional, so that, in fact, it is possible to omit most of the header section. The syntax defines a strict order for header entries and those that are present must follow this order.

Most entries are for documentation purposes and do not affect the meaning of the data in the rest of the file. However, the SDF version, hierarchy divider and time scale entries will, if present, change the way in which the file is interpreted.

Syntax

sdf_header ::= *sdf_version design_name*? *date*? *vendor*? *program_name*? *program_version*? *hierarchy_divider*? *voltage*? *process*? *temperature*? *time_scale*?

SDF Version Entry

The SDF version entry identifies the version of the Standard Delay Format specification to which the file conforms.

Syntax

sdf_version ::= **(SDFVERSION** QSTRING **)**

QSTRING is a character string, in double quotes. The first sub-string within QSTRING that matches one of the strings “1.0”, “2.0”, “2.1” or “3.0”, etc., identifies the SDF version. Other characters before and after this sub-string are permitted and should be ignored by readers when determining the SDF version.

Example

```
(SDFVERSION “OVI 3.0”)
```

In OVI SDF Version 2.1 and later, the SDF Version entry and QSTRING are required. In OVI SDF Version 1.0, the entry was required, but the QSTRING itself could be omitted. In OVI SDF Version 2.0, both the entry and the QSTRING were optional. Pre-OVI versions of SDF do not allow an SDF Version entry.

Writers of SDF files are recommended to include the SDF version entry, even in versions where it is optional. If this entry is present, the file should conform exactly to the syntax published for that SDF version.

Readers of SDF files may use the SDF version entry to adapt to the differences in file syntax between versions. If the file does not contain an SDF version entry, or one is present but the QSTRING field is blank, then the operation of the reader with regard to syntax differences is undefined and unexpected errors may result if the reader cannot automatically adapt to the syntax of the SDF version used.

Design Name Entry

The design name entry allows you to record in the SDF file the name of the design to which the timing data in the file applies. It is for documentation purposes and does not affect the meaning of the data in the file.

Syntax

design_name ::= **(DESIGN** QSTRING **)**

QSTRING is a name that identifies the design. Although this entry is not used by the annotator, it is recommended that, if it is included, it should be the name given to the top level of the design description. This is analogous to the **CELLTYPE** entry, and, in fact, the same name would be used in a cell entry for the entire design (for example, to carry all interconnect delay data). It should not be the instance name of the design in a test-bench; this would rather be used as part of the cell instance path in the **INSTANCE** entries for all cells.

Date Entry

The date entry allows you to record in the SDF file an indication of the currency of the data in the file. It is for documentation purposes and does not affect the meaning of the data in the file.

Syntax

date ::= **(DATE** QSTRING **)**

QSTRING is a character string, in double quotes, that represents the date and/or time when the data in the SDF file was generated.

Example

```
(DATE “Friday, September 17, 1993 - 7:30 p.m.”)
```

Vendor Entry

The vendor entry allows you to record in the SDF file the name of the company manufacturing the device to which the data in the file applies or who originated the program that created the file. It is for documentation purposes and does not affect the meaning of the data in the file.

Syntax

vendor ::= **(VENDOR** QSTRING **)**

QSTRING is a character string, in double quotes, containing the name of the vendor whose tools generated the SDF file.

Example

```
(VENDOR “Acme Semiconductor”)
```

Program Name Entry

The program name entry allows you to record in the SDF file the name of the program that created the file. It is for documentation purposes and does not affect the meaning of the data in the file.

Syntax

program_name ::= **(PROGRAM** QSTRING **)**

QSTRING is a character string, in double quotes, containing the name of the program used to generate the SDF file.

Example

```
(PROGRAM “timcalc”)
```

Program Version Entry

The program version entry allows you to record in the SDF file the version of the program that created the file. It is for documentation purposes and does not affect the meaning of the data in the file.

Syntax

program_version ::= **(VERSION** QSTRING **)**

QSTRING is a character string, in double quotes, containing the program version number used to generate the SDF file.

Example

```
(VERSION “version 1.3”)
```

Hierarchy Divider Entry

The hierarchy divider entry specifies which of the two permissible characters are used in the file to separate elements of a hierarchical path.

Syntax

hierarchy_divider ::= **(DIVIDER** HCHAR **)**

HCHAR is either a period (.), or a slash (/). It should not be in quotes.

Example

```
 (DIVIDER /)
. . .
   (INSTANCE a/b/c)
. . .
```

In this example, the hierarchy divider is specified to be the slash (/) character and hierarchical paths use / (rather than .) to separate elements.

If the SDF file does not contain a hierarchy divider entry then the default hierarchy divider is the period (.). See also the descriptions of IDENTIFIER and PATH in “Syntax Conventions” on page 4-2.

Voltage Entry

The voltage entry allows you to record in the SDF file the operating voltage or voltages for which the data in the file was computed. It is for documentation purposes and does not affect the meaning of the data in the SDF file.

Syntax

voltage ::= **(VOLTAGE** *rtriple* **)**
||= **(VOLTAGE** RNUMBER **)**

rtriple or RNUMBER indicates the operating voltage (in volts) at which the design timing was calculated or the constraints are to apply.

Example

```
(VOLTAGE 5.5:5.0:4.5)
```

Although this entry is not used by the annotator, it should be borne in mind that the order of delay and timing check limit values in *triple*s is minimum:typical:maximum. Since minimum delays usually occur at the highest supply voltage, it is more consistent with the use of *triple*s elsewhere in the file if the highest voltage is first in the voltage entry and the lowest voltage last.

Process Entry

The process entry allows you to record in the SDF file the process factor for which the data in the file was computed. It is for documentation purposes and does not affect the meaning of the data in the file.

Syntax

process ::= **(PROCESS** QSTRING **)**

QSTRING is a character string, in double quotes, which specifies the process operating envelope.

Example

```
(PROCESS “best=0.65:nom=1.0:worst=1.8”)
```

Temperature Entry

The temperature entry allows you to record in the SDF file the operating temperature for which the data in the file was computed. It is for documentation purposes and does not affect the meaning of the data in the file.

Syntax

temperature ::= **(TEMPERATURE** *rtriple* **)**
||= **(TEMPERATURE** RNUMBER **)**

rtriple or RNUMBER indicates the operating ambient temperature(s) of the design in degrees Celsius (centigrade).

Example

```
     (TEMPERATURE -25.0:25.0:85.0)
```

Timescale Entry

The timescale entry allows you to specify the units which you are using for all time values in the SDF file.

Syntax

time_scale ::= **(TIMESCALE** TSVALUE **)**

TSVALUE is a number followed by a unit. The number can be 1, 10, 100, 1.0, 10.0 or 100.0. The unit can be us, ns or ps representing microseconds, nanoseconds and picoseconds, respectively. A space may optionally separate the number and the unit. TSVALUE should not be in quotes.

Example

```
     (TIMESCALE 100 ps)
    . . .
         (IOPATH (posedge clk) q (2:3:4) (5:6:7))
    . . .
```

This example indicates that all time values in the file are to be multiplied by 100 picoseconds. Thus, the values supplied in the **IOPATH** entry are (0.2ns:0.3ns:0.4ns) and (0.5ns:0.6ns:0.7ns).

If the SDF file does not contain a timescale entry then all time values in the file will be assumed to be in nanoseconds. This has the same effect as a timescale entry of 1ns.

Cell Entries

A cell entry identifies a particular "region" or "scope" within a design and contains timing data to be applied there. For example, a cell entry might identify an unique occurrence of an ASIC physical primitive, such as a 2-input NAND gate, in the design and provide values for its timing properties, such as the input-to-output path delays. As well as identifying such design-specific regions, cell entries can identify all occurrences of a particular ASIC library physical primitive, such as a certain type of gate or flip-flop. Data is applied to all such library-specific regions in the design.

Syntax

cell ::= **(CELL** *celltype cell_instance timing_spec** **)**

The *celltype* and *cell_instance* fields identify a region of the design. The *timing_spec* field contains the timing data. These will be discussed in detail below.

Example

```
(CELL
  (CELLTYPE "DFF")
  (INSTANCE a/b/c)
  (DELAY
    (ABSOLUTE
      (IOPATH (posedge clk) q (2:3:4) (5:6:7) )
    )
  )
)
```

An SDF file may contain any number of cell entries (other than zero). The order of the cell entries is significant only if they have overlapping effect, in other words, if data from two different cell entries applies to the same timing property in the design. In this situation, the cell entries are processed strictly from the beginning of the file towards the end and the data they contain is applied in sequence to whatever region is appropriate to that cell entry. Where data is applied to a timing property previously referenced by the same SDF file, the new data will be applied over the old and the final value will be the cumulative effect, whether the data is applied as a replacement for the old value (absolute delays and timing checks) or is added to it (incremental delays).

Cell Type Entry

The **CELLTYPE** entry indicates the name of the cell.

Syntax

celltype ::= **(CELLTYPE** QSTRING **)**

QSTRING is a character string, in double quotes. If the region of the design identified by this cell entry is an occurrence of a physical primitive from an ASIC library, then QSTRING should be the name by which the cell is known in the library.

Example

```
        (CELLTYPE “DFF”)
```

In this example, the cell entry identifies an occurrence of a cell which has the name “DFF” (perhaps a D-type flip-flop).

If the cell entry identifies a region of the design which is a user-created level in the hierarchy, or, for example, the very top level, then QSTRING should be the user-created name for that part of the design.

Example

```
        (CELLTYPE “TOP”)
```

In this example, the cell entry identifies a user-created design block which the user has named “TOP”.

If the cell entry identifies a modeling primitive, in other words something that is not part of the physical design but is part of the implementation of a model in a particular analysis tool, then QSTRING should be the name by which the modeling primitive is known in that tool.

Example

```
        (CELLTYPE “buf”)
```

In this example, the cell entry identifies a “buf” modeling primitive in an analysis tool, perhaps a buf “gate” in a Verilog model.

Cell Instance Entry

The cell instance entry identifies the region or scope of the design for which the cell entry contains timing data. The name by which this region is known in the design must be consistent with the **CELLTYPE** entry for the cell. If the annotator locates the region and finds that its name does not match the **CELLTYPE** entry, it should indicate an error.

Syntax

cell_instance	::=	**(INSTANCE** PATH? **)**	
	||=	**(INSTANCE** WILDCARD **)**	
WILDCARD	::=	*****	// the asterisk character

The first form of the cell instance entry identifies an unique occurrence in the design of the region named in the cell type entry. If, for example, the cell is a physical primitive from an ASIC library, then a single occurrence of that cell on the chip will be identified. To do this, the cell instance entry provides a complete path through the design hierarchy to the cell or region of interest.

The hierarchical path must start at the level in the design at which the annotator will be instructed to apply the SDF file. Frequently, this is the topmost level. The path is extended down through the hierarchy by adding further levels to PATH.

Example

```
(CELL
  (CELLTYPE "DFF")
  (INSTANCE a1.b1.c1)
   . . .
)
```

In the above example, the complete hierarchical path is specified as `a1.b1.c1` following the **INSTANCE** keyword. The region identified is cell/block `c1` within block `b1`, which is in turn within block `a1`. The SDF file must be applied at the level containing `a1`. The period character separates levels or elements of the path. The example assumes that the hierarchy divider entry in the SDF header specified the hierarchy divider as the period character or, since period is the default, the entry was absent.

The timing data in the timing specifications of this cell entry apply only to the identified region of the design. If you do not specify PATH, i.e. you leave it blank, the default is the region (hierarchical level) in the design at which the annotator is instructed to apply the SDF file (see “The Annotator” page 3 in chapter 2). This can be useful for gathering all interconnect information into a top-level cell entry.

The second form of the cell instance entry can be used to associate timing data with all occurrences of the specified cell type. Instead of a hierarchical path, specify the wildcard character (*) after the **INSTANCE** keyword, as shown below.

Example

```
(CELL
  (CELLTYPE "DFF")
  (INSTANCE *)
   . . .
)
```

The effect of this cell instance entry will be to apply the timing data in this cell entry to all occurrences of the cell specified in the cell type entry. In this particular example, every DFF cell will receive the timing data. Note, however, that only cells contained within the region to which the annotator is instructed to apply the SDF file will be affected.

Cell entries using the wildcard cell instance specification are processed in sequence just like any other cell entry. No special action is taken to consolidate data in this cell entry with cell entries with the same cell type earlier or later in the file.

Timing Specifications

Each cell entry in the SDF file includes zero or more timing specifications which contain the actual timing data associated with that cell entry. There are three types of timing specifications that are identified by the **DELAY**, **TIMINGCHECK**, and **TIMINGENV** keywords.

Syntax

```
timing_spec ::= del_spec
            ||= tc_spec
            ||= te_spec
   del_spec ::= ( DELAY deltype+ )
    tc_spec ::= ( TIMINGCHECK tchk_def+ )
    te_spec ::= ( TIMINGENV te_def+ )
```

The **DELAY** keyword introduces delay entries which contain delay data and narrow-pulse propagation data for back-annotation.

Delay entries are described in the following section.

The **TIMINGCHECK** keyword introduces timing check entries which contain timing check limit data for back-annotation.

Timing check entries are described in “Timing Check Entries” on page 26.

The **TIMINGENV** keyword introduces timing environment entries which contain timing environment and constraint data for forward-annotation.

Timing environment entries are described in “Timing Environment Entries” on page 36.

Any number of delay entries, timing check entries and timing environment entries may be contained in a cell entry and they can occur in any order. However, it is preferable, for efficiency reasons, to put all delay and pulse propagation data in a single delay entry, all timing check data in a single timing check entry and all timing environment and constraint data in a single timing environment entry for each cell.

Delay Entries

Timing specifications that start with the **DELAY** keyword associate delay values with input-to-output paths, input ports, interconnects, and device outputs. They can also provide narrow-pulse propagation data for input-to-output paths.

Syntax

del_spec ::= **(DELAY** *deltype*+ **)**

Any number of *deltype* entries may appear in a *del_spec* entry. Each *deltype* will be a **PATHPULSE** or **PATHPULSEPERCENT** entry, specifying how pulses will propagate across paths in this cell, or **ABSOLUTE** or **INCREMENT** delay definition entries, containing delay values to be applied to the region identified by the cell.

Syntax

```
deltype  ::=  ( PATHPULSE input_output_path? value value? )
         ||=  ( PATHPULSEPERCENT input_ouput_path? value value? )
         ||=  ( ABSOLUTE del_def+ )
         ||=  ( INCREMENT del_def+ )
```

The following sections describe the *deltype* entries.

PATHPULSE

The **PATHPULSE** entry represents narrow-pulse propagation limits associated with a legal path between an input port and an output port of a device. These limits determine whether a pulse of a certain width can pass through the device and appear at the output.

Syntax

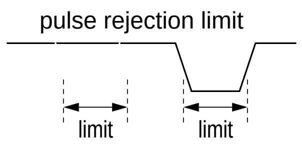
(PATHPULSE *input_output_path*? *value value*? **)**

input_output_path ::= *port_instance port_instance*

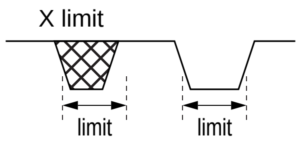
The first *port_instance* of *input_output_path* is an input or a bidirectional port.

The second *port_instance* of *input_ouput_path* is an output or a bidirectional port.

If *input_output_path* is omitted, then the data supplied refers to all input-to-output paths in the region identified by the cell entry. The annotator must locate all paths that are able to model narrow-pulse propagation in the applicable timing model and apply the supplied data.



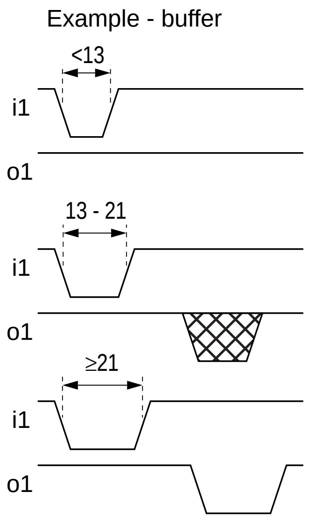
The first *value*, in time units, is the pulse rejection limit. This limit defines the narrowest pulse that can appear at the output port of the specified path. Any narrower pulse does not appear at the output.



The second *value*, in time units, is the X limit. This limit defines the minimum pulse width necessary to drive the output of the specified path to a known state; a narrower pulse causes the output to enter the unknown (X) state or is rejected (if smaller than the pulse rejection limit). Note that the X limit must be greater than the pulse rejection limit to carry any significance.

If you specify only one *value*, both limits are set to that value. In all cases *value* can be either a single number or a *triple*, but must not be negative.

Example

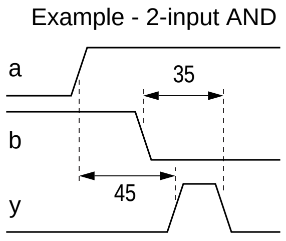


```
(INSTANCE x)
(DELAY
  (PATHPULSE i1 o1 (13) (21))
)
```

In this example of a simple buffer cell, the pulse rejection limit is specified as 13 time units and the X limit is specified as 21 time units. It is assumed that the high-to-low and low-to-high delays from i1 to o1 are the same. The first pulse, being shorter than 13, is rejected. The second pulse, being at least 13, but shorter than 21, appears at the output as an X. The third pulse, being at least 21, is passed to the output.

When narrow pulses arrive at an output due to changes at different inputs (rather than two changes at the same input, as in the above example), the two paths from the inputs to the output may have different limits. The assumption made in SDF is that the analysis tool will use the data for the path that terminated the pulse to control the pulse's appearance at the output.

Example



```
(INSTANCE x)
(DELAY
  (ABSOLUTE
    (IOPATH a y (45) (37))
    (IOPATH b y (43) (35))
  )
  (PATHPULSE a y (13) (24))
  (PATHPULSE b y (15) (21))
)
```

This much more complex example is for a 2-input AND gate where both inputs are high for a short period of time as one goes high just before the other goes low. Because of differences in the path delays from a to y and from b to y, the pulse that arrives at the output is 10 time units shorter than the overlap of the high states at a and b. The path from b to y is the one that caused the pulse to terminate. The analysis tool should reject this pulse if it is shorter than 15 and change the pulse to the X state if it is at least 15 but shorter than 21.

Note that the order in which the inputs changed is of no consequence; pulse propagation is controlled by the data associated with the path through which the transition propagates that ends the output pulse.

If a path has not been given data for its pulse rejection or X limits, then the analysis tool assumes a pulse rejection limit and an X limit equal to the path delay. Thus, if this path terminates a narrow pulse, the pulse will be rejected if it is shorter than the path delay or otherwise passed.

PATHPULSEPERCENT

The **PATHPULSEPERCENT** entry is the same as **PATHPULSE** but the values are expressed as a percentage (%) of the cell path delay from the input to the output.

Syntax

(PATHPULSEPERCENT *input_output_path*? *value value*? **)**

Neither *value* should be greater than 100.0. To have any effect, the second *value* (X limit) must be greater than the first *value* (pulse rejection limit).

Example

```
(INSTANCE x)
(DELAY
  (ABSOLUTE
    (IOPATH a y (45) (37))
  )
  (PATHPULSEPERCENT a y (25) (35))
)
```

In this example, the pulse rejection limit is specified as 25% of the delay time from `a` to `y` and the X limit is specified as 35% of this delay. If more than one *delval* is specified in the *delval_list* of an **IOPATH** entry, the analysis tool selects that corresponding to the transition than ended the pulse. So, for a high-going output pulse, which ends with a high-to-low transition, the percentages are applied to the high-to-low delay of the path. In the above example, where the high-to-low delay is 37, the pulse rejection limit is 25% of 37 and the X limit is 35% of 37. The data used for pulse control comes from the path that caused the pulse to terminate (in the same way as for the **PATHPULSE** construct).

Note that if the analysis tool is able to model narrow-pulse propagation with different limits for each output transition, the tool can pre-compute the limit values from the percentages and path delay values. The annotator, however, cannot do this as new values for path delays may be supplied after the **PATHPULSEPERCENT** entry is processed.

ABSOLUTE Delays

The **ABSOLUTE** keyword introduces delay data that replaces existing delay values in the design during annotation.

Syntax

(ABSOLUTE *del_def+*)

The delay definition entries, *del_def*, contain the actual data and describe where it belongs in the design.

Example

```
(CELL (CELLTYPE "DFF")
  (INSTANCE a.b.c)
  (DELAY
    (ABSOLUTE
      (IOPATH (posedge clk) q (22:28:33) (25:30:37))
      (PORT clr (32:39:49) (35:41:47))
    )
  )
)
```

Negative delay values can be specified for absolute delays to accommodate certain styles of ASIC cell characterization. However, note that not all analysis tools will be able to make sense of negative delays and some may set them to zero.

INCREMENT Delays

The **INCREMENT** keyword introduces delay data that is added to existing delay values in the design during annotation.

Syntax

(INCREMENT *del_def+* **)**

The delay definition entries, *del_def*, contain the actual data and describe where it belongs in the design. The same delay definition constructs are used for increment and absolute delays.

Example

```
(CELL (CELLTYPE "DFF")
  (INSTANCE a.b.c)
  (DELAY
    (INCREMENT
      (IOPATH (posedge clk) q (-4::2) (-7::5))
      (PORT clr (2:3:4) (5:6:7))
    )
  )
)
```

Negative delay values can be specified for increment delays, in which case, of course, the value existing in the design will be reduced. If any negative increment results in negative delays, note that not all analysis tools will be able to make sense of negative delays and may set them to zero.

Delay Definition Entries

Both absolute and increment delays are described by the same group of delay definition constructs.

Syntax

```
del_def ::= ( IOPATH port_spec port_instance
                    ( RETAIN delval_list )* delval_list )
        ||= ( COND QSTRING? conditional_port_expr
                    ( IOPATH port_spec port_instance
                    ( RETAIN delval_list )* delval_list ) )
        ||= ( CONDELSE
                    ( IOPATH port_spec port_instance
                    ( RETAIN delval_list )* delval_list ) )
        ||= ( PORT port_instance delval_list )
        ||= ( INTERCONNECT port_instance port_instance delval_list )
        ||= ( DEVICE port_instance? delval_list )
```

Specifying Delay Values

In the syntax descriptions above, you will see that each construct uses *delval_list* to specify the operating values to be applied. The section “Data Values” on page 4-7 provides a formal definition of *delval_list* along with related syntax constructs. However, here we discuss *delval_list* in the context of specifying delay and pulse control data for the various delay constructs in SDF.

The delay data in each delay definition entry is specified in a list of *delval*s.

Syntax

```
delval_list ::= delval
            ||= delval delval
            ||= delval delval delval
            ||= delval delval delval delval delval? delval?
            ||= delval delval delval delval delval delval
                      delval delval? delval? delval? delval? delval?
```

The number of *delval*s in the *delval_list* can be one, two, three, six or twelve. Note, however, that the amount of data you include in a delay definition entry must be consistent with the analysis tool’s ability to model that kind of delay. For example, if the modeling primitives of a particular tool can accept only three delay values, perhaps rising, falling and “Z” transitions, you should not attempt to annotate different values for 0→1 and Z→1 transitions or for 1→Z and 0→Z transitions. It is recommended that in such situations annotators combine the information given in some documented manner and issue a warning.

The following paragraphs define the semantics of *delval_list*s of various lengths.

If twelve *delval*s are specified in *delval_list*, then each corresponds, in sequence, to the delay value applicable when the port (for **IOPATH** and **INTERCONNECT**, the output port) makes the following transitions:

0→1, 1→0, 0→Z, Z→1, 1→Z, Z→0, 0→X, X→1, 1→X, X→0, X→Z, Z→X

If fewer than twelve *delval*s are specified in *delval_list*, then the table below shows how the delays for each transition of the port are found from the values given.

	Number of *delvals* in *delval_list*		
Transition	2	3	6
0→1	01	01	01
1→0	10	10	10
0→Z	01	–Z	0Z
Z→1	01	01	Z1
1→Z	10	–Z	1Z
Z→0	10	10	Z0
0→X	01	min(01,–Z)	min(01,0Z)
X→1	01	01	max(01,Z1)
1→X	10	min(10,–Z)	min(10,1Z)
X→0	10	10	max(10,Z0)
X→Z	max(01,10)	–Z	max(0Z,1Z)
Z→X	min(01,10)	min(01,10)	min(Z0,Z1)

If only two *delval*s are specified, the first (“rising”) is denoted in the table by 01 and the second (“falling”) by 10.

If three *delval*s are specified, the first and second are denoted as before and the third, the “Z” transition value, by –Z.

If six *delval*s are specified, they are denoted, in sequence, by 01, 10, 0Z, Z1, 1Z and Z0.

If a single *delval* is specified, it applies to all twelve possible transitions. This is not shown in the table.

In a *delval_list*, any *delval*s can be null, that is, the parentheses enclosing the RNUMBER or *rtriple* are empty (see “Data Values” on page 4-7). The meaning of this is the same as missing numbers in an *rtriple*: no data is supplied and values should not be changed by the annotator. Such null *delval*s act as “placeholders” to allow you to specify *delval*s further down the list.

Example

```
(IOPATH i3 o1 () () (2:4:5) (4:5:6) (2:4:5) (4:5:6))
```

In this example, 0→1 and 1→0 delay values are not specified and might not even be present in the timing model. A *delval_list* consisting of nothing but null *delval*s is permitted by the syntax and has no effect.

In *delval_list*s of length six and twelve, it is permissible to omit trailing null *delval*s. Thus, a list of four *delval*s, for example, provides data for the 0→1, 1→0, 0→Z and Z→1 transitions, but not for the 1→Z, Z→0 transitions. Note that omitting three *delval*s is going too far as a mapping is defined above for an *delval_list* of three *delval*s onto all six transitions.

Each *delval* is either an *rvalue* or a group of two or three *rvalue*s enclosed in parentheses.

Syntax

delval	::=	*rvalue*
	||=	**(** *rvalue rvalue* **)**
	||=	**(** *rvalue rvalue rvalue* **)**

When a single *rvalue* is used, it specifies the delay value. When two *rvalue*s in parentheses are used, the first *rvalue* specifies the delay, as if a single *rvalue* were given. The second specifies both the pulse rejection limit, or “r-limit”, associated with this delay, and the X-limit, or “e-limit”. When three *rvalues* are used, the first specifies the delay, the second specifies the pulse rejection limit, or “r-limit”, and the third specifies the X-limit, or “e-limit”. This allows pulse control data to be associated in a uniform way with all types of delays in SDF. Note that since any *rvalue* can be an empty pair of parentheses, each type of delay data can be annotated or omitted as the need arises.

Each *rvalue* is either a single RNUMBER or an *rtriple*, containing three RNUMBERs separated by colons, in parentheses.

Syntax

rvalue	::=	**(** RNUMBER? **)**
	||=	**(** *rtriple*? **)**

The use of single RNUMBERs and *rtriple*s should not be mixed in the same SDF file. All RNUMBERs can have negative, zero or positive values.

The use of triples in SDF allows you to carry three sets of data in the same file. Each number in the triple is an alternative value for the data and is typically selected from the triple by the annotator or analysis tool on an instruction from the user. The prevailing use of the three numbers is to represent minimum, typical and maximum values computed at three process/operating conditions for the entire design. Any one or any two (but not all three) of the numbers in a triple may be omitted if the separating colons are left in place. This indicates that no value has been computed for that data, and the annotator should not make any changes if that number is selected from the triple. For absolute delays, this is **not** the same as entering a value of 0.0.

The following sections describe delay definition entries.

Input/Output Path Delays

The **IOPATH** entry represents the delays on a legal path from an input/bidirectional port to an output/bidirectional port of a device. Each delay value is associated with a unique input port/output port pair.

Syntax

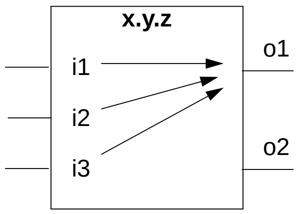
(IOPATH *port_spec port_instance delval_list* **)**

port_spec is an input or a bidirectional port and can have an edge identifier.

port_instance is an output or a bidirectional port. It cannot have an edge identifier. Delay data for the different transitions at the path output port are conveyed by supplying an ordered list of values as described above.

delval_list is the **IOPATH** delay data from *port_spec* to *port_instance*.

If the timing model includes conditions (state dependency) for the path delay between the two specified ports, the specified *delval* is still applied. If the model includes more than one delay path, each distinguished by its conditions, then the data applies to all of them. This has the same effect as specifying all paths (using the **COND** or **CONDELSE** keyword with **IOPATH** as described below) with the same **IOPATH** delay *delval_list*.



Example

```
(INSTANCE x.y.z)
(DELAY
  (ABSOLUTE
    (IOPATH (posedge i1) o1 (2:3:4) (4:5:6))
    (IOPATH i2 o1 (2:4:5) (5:6:7))
    (IOPATH i3 o1 () () (2:4:5) (4:5:6) (2:4:5) (4:5:6))
  )
)
```

Conditional Path Delays

The **COND** keyword allows the specification of conditional (state-dependent) input-to-output path delays.

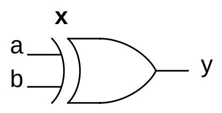
Syntax

(COND QSTRING? *conditional_port_expr*
        (IOPATH *port_spec port_instance delval_list* **))**

QSTRING is an optional symbolic name that can stand in for the expression itself for annotators that operate by matching named placeholders in the model to SDF constructs. See “Condition Labels”, below, for a full explanation.

conditional_port_expr is the description of the state dependency of the path delay. The syntax of *conditional_port_expr* is shown in “Conditions for Path Delays” on page 4-9. The perceptive reader will notice that this expression evaluates to a logical signal, rather than a boolean. The intent is that the analysis tool should treat a logical zero as FALSE and any other logical value (1, X or Z) as TRUE and that a particular conditional path delay in the timing model is used only if the condition is TRUE.

port_spec, *port_instance* and *delval_list* have exactly the same meaning as in **IOPATH** entries without the **COND** keyword as described above, except that the annotator must locate a path delay with a condition matching the one specified and apply the data only to that. Other path delays from the same input port to the same output port but with different conditions in the timing model will not receive the data. Annotators may differ in their capabilities to match a condition in SDF to conditions in the timing model. Where the analysis tool uses the same syntax as SDF (derived from the Verilog language), the annotator may require an exact character-for-character match in the string representations of the conditions.



Example

```
(INSTANCE x)
(DELAY
  (ABSOLUTE
    (COND b  (IOPATH a y (0.21) (0.54) ) )
    (COND ~b (IOPATH a y (0.27) (0.34) ) )
    (COND a  (IOPATH b y (0.42) (0.44) ) )
    (COND ~a (IOPATH b y (0.37) (0.45) ) )
  )
)
```

The **CONDELSE** keyword allows the specification of default delays for conditional paths. The default delay is the delay that will be in force if, during the simulation or analysis, none of the conditions specified for the path in the model are TRUE but a signal must still be propagated over the path.

Syntax

(CONDELSE (IOPATH *port_spec port_instance delval_list* **))**

This construct should be used only where the cell timing model includes an explicit mechanism for providing default delays. The annotator should match this SDF construct to such a mechanism in the model. It should not attempt to locate conditions for the path which have not been specified in **COND** constructs.

Condition Labels

Annotators may operate by mapping constructs in the SDF file into symbolic names, locating placeholders with those names in the models and applying values from the SDF file to the variables associated with those placeholders. (An example of this is the annotator for VITAL models in a VHDL simulator.) To ease the problem of mapping a *conditional_port_expr* construct (or the *timing_check_condition* construct in timing checks, later) into symbolic names, these can optionally be preceded by a QSTRING.

Clearly, for a tool that uses a name mapping annotation scheme, models must be constructed so as to contain the correct placeholders. Therefore, the mapping algorithm of the tool's annotator must be clearly documented and available to users. The description of the mapping must include the way in which the QSTRING is used in constructing the name. For example, it may be appended to a name constructed from other information in the SDF file such as the type of construct, port names, etc. The description should also explain what will happen if the QSTRING is absent in a conditional construct and what will happen in certain timing checks where two QSTRINGs are possible.

The intent of SDF is that the QSTRING should stand in place of the *conditional_port_expr* or *timing_check_condition* in constructing unique placeholder names for each state or condition in which a timing property might have a different annotated value.

Output Retain Delays

The **RETAIN** entry represents the time for which an output/bidirectional port will retain its previous logic value after a change at a related input/ bidirectional port. This is commonly used on paths from the address or select inputs to the data outputs of memory and register file circuits.

Syntax

(IOPATH *port_spec port_instance*
    (RETAIN *delval_list* **)*** delval_list**)**
(COND QSTRING? *conditional_port_expr*
    (IOPATH *port_spec port_instance* **(RETAIN** *delval_list* **)*** *delval_list* **))**
(CONDELSE
    (IOPATH *port_spec port_instance* **(RETAIN** *delval_list* **)*** *delval_list* **))**

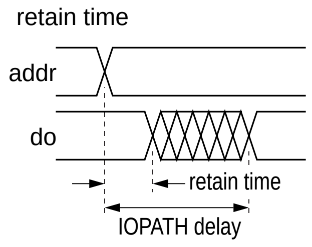
port_spec is an input or a bidirectional port and can have an edge identifier.

port_instance is an output or a bidirectional port. It cannot have an edge identifier. Delay data for the different transitions at the path output port are conveyed by supplying an ordered list of values as described above in "Specifying Delay Values" on page 16.

delval_list is the retain time data from *port_spec* to *port_instance*.

This construct should be used only where the cell timing model includes an explicit mechanism for providing retain times. The annotator should match this SDF construct to such a mechanism in the model.

The delays in *delval_list* for consecutive **RETAIN** statements must be strictly monotonically increasing.



Example

```
        (IOPATH addr[13:0] do[7:0]

          (RETAIN (4:5:7) (5:6:9))
```

In this example, the retain time of the bus `do[7:0]` with respect to changes on the bus `addr[7:0]` is described. It is assumed that the model for this cell contains path delays from `addr` to `do` and also a modeling construct to receive the retain times written so that after the retain time, `do` goes to the X state. The first *delval*, `(4:5:7)`, is the "rising" time and will be used for `do` going from low to X. The second *delval*, `(5:6:9)`, is the "falling" time and will be used for `do` going from high to X.

As with **IOPATH** entries, **RETAIN** entries can be made conditional or state dependent by the use of the **COND** and **CONDELSE** keywords.

Port Delays

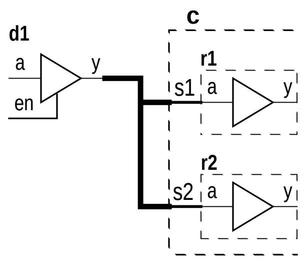
The **PORT** entry is for the specification of interconnect delays (actual or estimated) that are modeled as delay at input ports. The start point for the delay path (the driving output port) is not specified.

Syntax

(PORT *port_instance delval_list* **)**

port_instance is an input or bidirectional port.

delval_list is the **PORT** delay of the *port_instance*.



Example

```
(INSTANCE c)
(DELAY
  (ABSOLUTE
    (PORT r1.a (0.01:0.02:0.03))
    (PORT r2.a (0.03:0.04:0.05))
  )
)
```

Analysis tools must apply delay values obtained from SDF **PORT** entries before timing checks are applied. Thus, this construct models delay in the physical interconnect between the driver and the driven cell port.

Interconnect Delays

The **INTERCONNECT** entry is for the specification of interconnect delays (actual or estimated) that are modeled independently for each driver-to-driven path. Both start and end points for the delay path are specified.

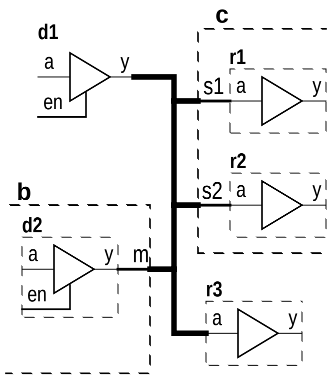
Syntax

(INTERCONNECT *port_instance port_instance delval_list* **)**

The first *port_instance* is an output or bidirectional port.

The second *port_instance* is an input or bidirectional port.

delval_list is the **INTERCONNECT** delay between the output and input ports.



Example

```
(INSTANCE top)
(DELAY
  (ABSOLUTE
    (INTERCONNECT  d1.y   c.r1.a (0.01:0.02:0.03))
    (INTERCONNECT  d1.y   c.r2.a (0.03:0.04:0.05))
    (INTERCONNECT  d1.y   r3.a   (0.05:0.06:0.07))
    (INTERCONNECT  b.d2.y c.r1.a (0.04:0.05:0.06))
    (INTERCONNECT  b.d2.y c.r2.a (0.02:0.03:0.04))
    (INTERCONNECT  b.d2.y r3.a   (0.02:0.03:0.04))
  )
)
```

Although **INTERCONNECT** entries are the most general way in which interconnect delays can be expressed, some analysis tools may not be able to model independent delay values over each driver-to-driven path on a net with more than one driver. Such tools may map **INTERCONNECT** entries into equivalent input port delays (such as would directly arise from **PORT** entries), sometimes losing information in the process. Even tools which can model independent delays over each path may do so less efficiently than input port delays. Writers of SDF files should bear this in mind when choosing whether to use **PORT** entries or **INTERCONNECT** entries or a combination of both to model interconnect delay.

Device Delays

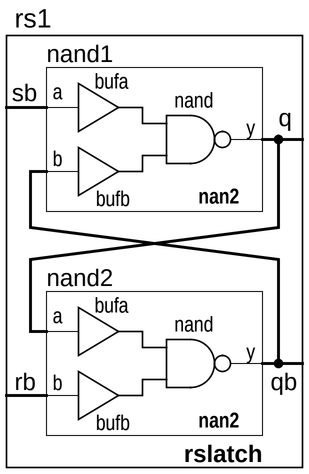
The **DEVICE** entry represents the delay of all paths through a cell to the specified output port. This construct is intended primarily for use with distributed timing models where the cell to which it is applied is a modeling primitive. If it is used at a higher level in the hierarchy, then the effect is to apply the delay data to all input-to-output paths across the cell that terminate at the specified port.

Syntax

(DEVICE *port_instance*? *delval_list* **)**

port_instance is optional and, if present, specifies the output port to which the delay data is to be applied. If a cell has more than one output, you can therefore include several **DEVICE** entries in a single **CELL** entry, each indicating the desired output port using *port_instance*, and attach different delay data to each output. If *port_instance* is omitted, all paths to all output ports of the region identified in the cell entry receive the same delay data.

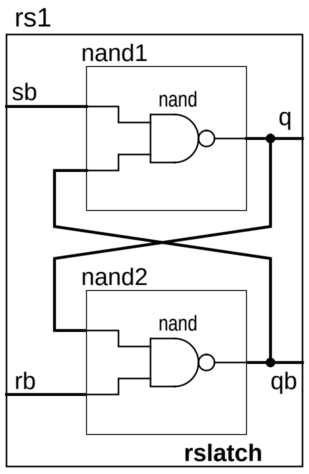
delval_list is the delay data. The number of *triple*s in *delval_list* must correspond to the capabilities of the modeling primitives of the target analysis tool. For example, Verilog “gates” can accept one, two, or in some cases, three delay values, but never six or twelve.



Example

```
(CELL
  (CELLTYPE “buf”)
  (INSTANCE rs1.nand1.bufa)
  (DELAY
    (ABSOLUTE
      (DEVICE (1:3:8) (4:5:7))
    )
  )
)
(CELL
  (CELLTYPE “buf”)
  (INSTANCE rs1.nand1.bufb)
  (DELAY
    (ABSOLUTE
      (DEVICE (2:4:9) (6:8:12))
    )
  )
)
```

In this example, a 2-input NAND gate model, `nan2`, is constructed in a distributed delay style from two buffer primitives, `bufa` and `bufb`, and a NAND gate primitive, `nand`. Two such NAND gates, `nand1` and `nand2`, are instantiated to create a design for an RS latch. This is then instantiated in a higher level of the design as `rs1`. The SDF file demonstrates the annotation of delays to the `a`-to-`y` and `b`-to-`y` paths through the top NAND gate. The first of these defines the input-to-output path delay from `sb` to `q` of the RS latch; the second contributes to the `rb` to `q` delay. The delay on `bufa` also contributes to the `sb`-to-`qb` delay.



Example

```
(CELL
  (CELLTYPE “rslatch“)
  (INSTANCE rs1)
  (DELAY
    (ABSOLUTE
      (DEVICE q (1:3:8) (4:5:7))
      (DEVICE qb (2:4:9) (6:8:12))
    )
  )
)
```

In this example, the same RS latch is described in a pin-to-pin modeling style. Two `nand` gate primitives are connected to form the functional part of the model and all timing information is described separately in a timing model of whatever form the analysis tool requires. Typically, this timing model would specify input-to-output delay paths from `sb` to `q`, `rb` to `q`, `sb` to `qb` and `rb` to `qb`. The above excerpt from an SDF file annotates values for all paths to the q and qb outputs. It will have exactly the same effect as the following:

```
(CELL
  (CELLTYPE “rslatch“)
  (INSTANCE rs1)
  (DELAY
    (ABSOLUTE
      (IOPATH sb q (1:3:8) (4:5:7))
      (IOPATH rb q (1:3:8) (4:5:7))
      (IOPATH sb qb (2:4:9) (6:8:12))
      (IOPATH rb qb (2:4:9) (6:8:12))
    )
  )
)
```

Timing Check Entries

Timing specifications that start with the **TIMINGCHECK** keyword associate timing check limit values with specific cell instances.

Syntax

tc_spec ::= **(TIMINGCHECK** *tchk_def+* **)**

Any number of *tchk_def* entries may appear in a *tc_spec* entry. Each *tchk_def* will be a **SETUP**, **HOLD**, **SETUPHOLD**, **RECOVERY**, **REMOVAL**, **RECREM**, **SKEW**, **WIDTH**, **PERIOD** or **NOCHANGE** timing check entry, containing timing check limit values for this cell entry.

Timing Checks

Timing check entries specify limits in the way in which a signal can change or two signals can change in relation to each other for reliable circuit operation. EDA analysis tools use this information in different ways:

- Simulation tools issue warnings about signal transitions that violate timing checks.
- Timing analysis tools identify delay paths that might cause timing check violations and may determine the constraints for those paths.

Syntax

```
tchk_def ::= ( SETUP port_tchk port_tchk value )
         ||= ( HOLD port_tchk port_tchk value )
         ||= ( SETUPHOLD port_tchk port_tchk rvalue rvalue )
         ||= ( SETUPHOLD port_spec port_spec rvalue rvalue scond? ccond? )
         ||= ( RECOVERY port_tchk port_tchk value )
         ||= ( REMOVAL port_tchk port_tchk value )
         ||= ( RECREM port_tchk port_tchk rvalue rvalue )
         ||= ( RECREM port_spec port_spec rvalue rvalue scond? ccond? )
         ||= ( SKEW port_tchk port_tchk rvalue )
         ||= ( WIDTH port_tchk value )
         ||= ( PERIOD port_tchk value )
         ||= ( NOCHANGE port_tchk port_tchk rvalue rvalue )
```

Conditional Timing Checks

The **COND** keyword allows the specification of conditional timing checks. Its use is rather different from the specification of conditional input-output path delays described in “Conditional Path Delays” on page 19 in that the condition is associated with the specification of a port rather than the entry as a whole.

Syntax

```
port_tchk ::= port_spec
          ||= ( COND QSTRING? timing_check_condition port_spec )
```

QSTRING is an optional symbolic name that can stand in for the expression itself for annotators that operate by matching named placeholders in the model to SDF constructs. See "Condition Labels" on page 20, for a full explanation.

timing_check_condition is the description of the state dependency of the timing check. The syntax of *timing_check_condition* is shown in "Conditions for Timing Checks" on page 4-9. The perceptive reader will notice that this expression evaluates to a logical signal, rather than a boolean. The intent is that the analysis tool should treat a logical zero as FALSE and any other logical value (1, X or Z) as TRUE and that a particular conditional timing check in the timing model is used only if the condition is TRUE.

The annotator must locate in the timing model a timing check with conditions matching those specified. Other timing checks of the same kind but with different conditions from the SDF entry will not receive the data. SDF timing check entries with no conditions match any timing check in the model of the same kind and between the ports specified in the SDF entry.

An alternative syntax is available for **SETUPHOLD** and **RECREM** timing checks. This associates the conditions with the "stamp" and "check" events in the analysis tool rather than the *port_spec*. Separate conditions can be supplied for the "stamp" and "check" events using the **SCOND** and **CCOND** keywords. Note, **SCOND** or **CCOND** or both **SCOND** and **CCOND** take precedence over **COND**.

Syntax

scond ::= **(SCOND** QSTRING? *timing_check_condition* **)**
ccond ::= **(CCOND** QSTRING? *timing_check_condition* **)**

For the setup phase of a setuphold timing check, the "stamp" condition applies to the data port and the "check" condition to the clock or gate port. For the hold phase, the "stamp" condition applies to the clock or gate port and the "check" condition to the data port.

These conditions restore flexibility in expressing conditions that is lost when **SETUP** and **HOLD** are combined into **SETUPHOLD**, or when **RECOVERY** and **REMOVAL** are combined into **RECREM**. For example, here are separate **SETUP** and **HOLD** statements for the same clock and data signals, but with the condition attached to the clock in one case, and to the data in the other:

```
(SETUP d (COND enb clk) (5))
(HOLD  (COND enb d) clk (7))
```

These conditions cannot be combined into a single **SETUPHOLD** as shown here:

```
        (SETUPHOLD (COND enb d) (COND enb clk) (5) (7))
```

This is because there is no way to specify that the condition should only apply to signal clk for SETUP checks, and only to signal d for HOLD checks. The SCOND and CCOND fields provide this capability. By definition, the CCOND field defines a condition for the check event (the 2nd event):

```
        (SETUPHOLD d clk (5) (7) (CCOND enb))
```

Edge Specifications

Any *port_spec* can be qualified with an edge identifier as follows:

Syntax

port_spec	::=	*port_instance*
	||=	*port_edge*
port_edge	::=	**(** EDGE_IDENTIFIER *port_instance* **)**

This will be termed an "edge specification". When the annotator is locating a timing check at specified ports in the timing model, it must match the edge specification as well as the port names. A port without an edge specification in SDF matches any edge specification in the model.

Example

```
   (CELL (CELLTYPE "DFF")
     (INSTANCE a.b.c)
     (TIMINGCHECK
       (SETUP din (posedge clk) (3:4:5.5))
       (HOLD din (posedge clk) (4:5.5:7))
     )
   )
```

This example shows a cell entry which provides values for setup and hold timing checks with respect to the rising edge of the clock signal.

Specifying Timing Check Limit Values

In the syntax descriptions of the timing check constructs, you will see that either *rvalue* or *value* is used to specify the timing check limit to be applied. Although *rvalue* may be negative, *value* must be zero or positive.

Each *rvalue* or *value* may be a single value (RNUMBER or NUMBER, respectively) or three values separated by colons, (an *rtriple* or *triple*) representing three sets of data for minimum, typical and maximum delay conditions. However, the use of single RNUMBER/NUMBERs and *rtriple*/*triple*s should not be mixed in the same SDF file.

The use of triples in SDF allows you to carry three sets of data in the same file. Each number in the triple is an alternative value for the data and is typically selected from the triple by the annotator or analysis tool on an instruction from the user. The prevailing use of the three numbers is to represent minimum, typical and maximum values computed at three process/operating conditions for the entire design. Any one or any two (but not all three) of the numbers in a triple may be omitted if the separating colons are left in place. This indicates that no value has been computed for that data, and the annotator should not make any changes if that number is selected from the triple.

SETUPHOLD, **RECREM** and **NOCHANGE** timing checks have two *rvalue*s, the first for the setup limit and the second for the hold limit.

Setup Timing Check

The **SETUP** entry specifies limit values for a setup timing check.

Setup and hold timing checks are used to define a time interval during which a "data" signal must remain stable in order for a transition of a "clock" or "gate" signal to store the data successfully in a storage device (flip-flop or latch). The setup time limit defines the part of the interval before the clock transition; the hold time limit defines the part of the interval after the clock transition. Any change to the data signal within this interval results in a timing violation. To shift the interval with respect to the clock transition, either the setup time or the hold time can be negative; however, their sum must always be greater than zero.

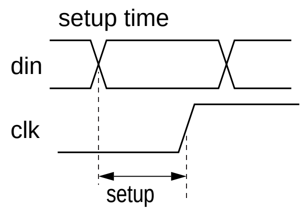
Syntax

(SETUP *port_tchk port_tchk value* **)**

The first *port_tchk* identifies the data port. If it includes an edge specification, then the data is for a setup time check with respect only to the specified transition at the data port.

The second *port_tchk* identifies the clock/gate port and will normally include an edge specification to identify the active edge of the clock or the active-to-inactive transition of the gate.

value is the **SETUP** time limit between the data and clock ports and must not be negative.



Example

```
(INSTANCE x.a)
(TIMINGCHECK
  (SETUP din (posedge clk) (12))
)
```

As with all *port_tchk*s, the **COND** construct can be used to specify conditions associated with the setup timing check.

Hold Timing Check

The **HOLD** entry specifies limit values for a hold timing check.

Syntax

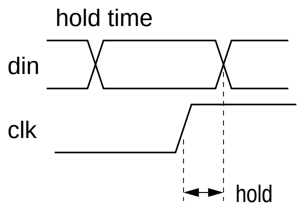
(HOLD *port_tchk port_tchk value* **)**

The first *port_tchk* identifies the data port.

The second *port_tchk* identifies the clock port.

value is the **HOLD** time between the data and clock events and must not be negative.

See "Setup Timing Check" above for a description of the use of hold timing checks and more information about the use of edge specifications in this context.



Example

```
(INSTANCE x.a)
(TIMINGCHECK
  (HOLD din (posedge clk) (9.5))
  . . .
)
```

As with all *port_tchk*s, the **COND** construct can be used to specify conditions associated with the hold timing check.

SetupHold Timing Check

The **SETUPHOLD** entry specifies setup and hold limits in a single entry.

Syntax

(SETUPHOLD *port_tchk port_tchk rvalue rvalue*)
(SETUPHOLD *port_spec port_spec rvalue rvalue scond*? *ccond*? **)**

The first *port_tchk* or *port_spec* identifies the data port.

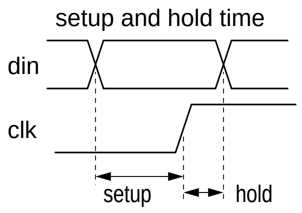
The second *port_tchk* or *port_spec* identifies the clock port.

As with all *port_tchk*s, the **COND** construct can be used in the first form of the setuphold timing check to specify conditions associated with the ports.

The first *rvalue* is the setup time and the second *rvalue* is the hold time. Either can be negative, however their sum must be greater than zero.

In the second syntax form, *scond* and *ccond* are the "stamp" and "check" conditions as described above in "Conditional Timing Checks" on page 26.

See "Setup Timing Check" above for the use of setup and hold timing checks and edge specifications in this context.



Example

```
(INSTANCE x.a)
(TIMINGCHECK
  (SETUPHOLD (COND ~reset din) (posedge clk) (12) (9.5))
)
```

This SDF entry will match setup and hold timing checks in the model that are conditional on `~reset` at the time the `din` port changes. At this time in the analysis tool, `~reset` must evaluate to TRUE, i.e. the `reset` signal must be in the zero, X or Z states, for the checks to be performed.

Example

```
(INSTANCE x.a)
(TIMINGCHECK
  (SETUPHOLD din (posedge clk) (12) (9.5) (CCOND ~reset))
)
```

This SDF entry, using the second syntax form, will match setup and hold timing checks in the model that are conditional on `~reset` at the time of the "check" event. For the setup phase of the check, this will be when the `clk` port undergoes a **posedge** transition. For the hold phase of the check, this will be when the `din` port undergoes any transition.

Recovery Timing Check

The **RECOVERY** entry specifies limit values for recovery timing checks. A recovery timing check is a limit of the time between the release of an asynchronous control signal from the active state and the next active clock edge, for example between clearbar and the clock for a flip-flop. If the active edge of the clock occurs too soon after the release of the clearbar, the state of the flip-flop will become uncertain — it could be the value set by the clearbar, or it could be the value clocked into the flip-flop from the data input. In other respects, a recovery check is similar to a setup check.

Syntax

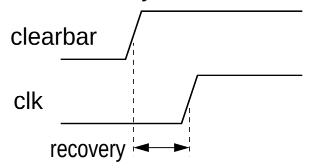
(RECOVERY *port_tchk port_tchk value* **)**

The first *port_tchk* refers to the asynchronous control signal and will normally have an edge identifier associated with it to indicate which transition corresponds to the release from the active state.

The second *port_tchk* refers to the clock (flip-flops) or gate (latches). This will also normally have an edge identifier to indicate the active edge of the clock or the closing edge of the gate.

value is the recovery limit value and must not be negative. It is the time it takes a device to recover after an extraordinary operation, such as set or reset, so that it can reliably return to normal operation, such as clocking in of new data.

recovery time



Example

```
(INSTANCE x.b)
(TIMINGCHECK
  (RECOVERY (posedge clearbar) (posedge clk) (11.5))
)
```

As with all *port_tchk*s, the **COND** construct can be used to specify conditions associated with the recovery timing check.

Removal Timing Check

The **REMOVAL** entry specifies limit values for removal timing checks. A removal timing check is a limit of the time between an active clock edge and the release of an asynchronous control signal from the active state, for example between the clock and the clearbar for a flip-flop. If the release of the clearbar occurs too soon after the active edge of the clock, the state of the flip-flop will become uncertain — it could be the value set by the clearbar, or it could be the value clocked into the flip-flop from the data input. In other respects, a removal check is similar to a hold check.

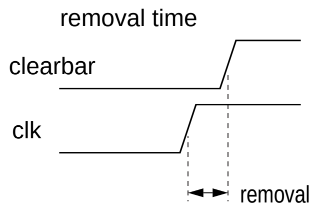
Syntax

(REMOVAL *port_tchk port_tchk value* **)**

The first *port_tchk* refers to the asynchronous control signal and will normally have an edge identifier associated with it to indicate which transition corresponds to the release from the active state.

The second *port_tchk* refers to the clock (flip-flops) or gate (latches). This will also normally have an edge identifier to indicate the active edge of the clock or the closing edge of the gate.

value is the removal limit value and must not be negative. It is the time for which an extraordinary operation, such as set or reset, must persist to insure that a device will ignore any normal operation, such as clocking in of new data.



Example

```
(INSTANCE x.b)
(TIMINGCHECK
  (REMOVAL (posedge clearbar) (posedge clk) (6.3))
)
```

As with all *port_tchk*s, the **COND** construct can be used to specify conditions associated with the recovery timing check.

Recovery/Removal Timing Check

The **RECREM** construct specifies both recovery and removal limits in a single entry.

Syntax

(RECREM *port_tchk port_tchk rvalue rvalue*)
(RECREM *port_spec port_spec rvalue rvalue scond*? *ccond*? **)**

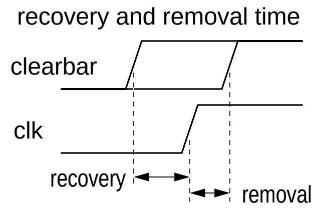
The first *port_tchk* or *port_spec* identifies the asynchronous control port.

The second *port_tchk* or *port_spec* identifies the clock (for flip-flops) or gate (for latches) port.

As with all *port_tchk*s, the **COND** construct can be used in the first form of the recovery/removal timing check to specify conditions associated with the ports.

The first *rvalue* is the recovery time and the second *rvalue* is the removal time. Either can be negative, however their sum must be greater than zero.

In the second syntax form, *scond* and *ccond* are the “stamp” and “check” conditions as described above in “Conditional Timing Checks” on page 26.



Example

```
(INSTANCE x.b)
(TIMINGCHECK
  (RECREM (posedge clearbar) (posedge clk) (1.5) (0.8))
)
```

This example specifies a recovery time of 1.5 and a removal time of 0.8. The recovery time limit (1.5 time units) defines the part of the interval before the clock transition; the removal time limit (0.8 time units) defines the part of the interval after the clock transition. Any change to the clearbar signal within this interval results in a timing violation.

Skew Timing Check

The **SKEW** entry specifies limit values for signal skew timing checks. A signal skew limit is the maximum allowable delay between two signals, which if exceeded causes devices to behave unreliably.

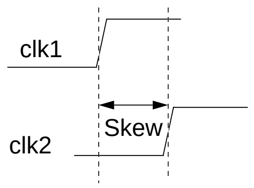
Syntax

(SKEW *port_tchk port_tchk rvalue* **)**

The first *port_tchk* is the stamp event and can include an edge specification.

The second *port_tchk* is the check event and can include an edge specification.

rvalue is the maximum skew limit.



Example

```
(INSTANCE x)
(TIMINGCHECK
  (SKEW (posedge clk1) (posedge clk2) (6))
)
```

As with all *port_tchk*s, the **COND** construct can be used to specify conditions associated with the skew timing check.

Width Timing Check

The **WIDTH** entry specifies limits for a minimum pulse width timing check. The minimum pulse width timing check is the minimum allowable time for the positive (high) or negative (low) phase of each cycle. If a signal has unequal phases, you can specify a separate width check for each phase.

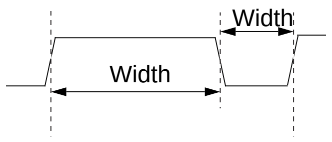
Syntax

(**WIDTH** *port_tchk value* **)**

port_tchk refers to the port at which the minimum pulse width timing check is applied. If it includes an edge specification, then the data will apply to the width check for the phase of the signal beginning with this edge (see example below). If *port_tchk* does not include an edge specification, then the data applies to both high and low phases of the signal.

value is the minimum pulse width limit and cannot be negative.

Example



```
(INSTANCE x.b)
(TIMINGCHECK
  (WIDTH (posedge clk) (30))
  (WIDTH (negedge clk) (16.5))
)
```

In this example, the first minimum pulse width check is for the phase beginning with the positive clock edge, i.e. the high phase of the clock, and the second minimum pulse width check is for the phase beginning with the negative clock edge, i.e. the low phase.

As with all *port_tchk*s, the **COND** construct can be used to specify conditions associated with the minimum pulse width timing check.

Period Timing Check

The **PERIOD** entry specifies limit values for a minimum period timing check. The minimum period timing check is the minimum allowable time for one complete cycle of the signal.

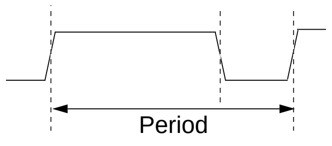
Syntax

(**PERIOD** *port_tchk value* **)**

port_tchk refers to the port at which the minimum period timing check is applied. If it includes an edge specification, then the data will apply to the period check between consecutive edges of this direction (see example below). If *port_tchk* does not include an edge specification, then the data applies both to period checks between consecutive rising edges and between consecutive falling edges if they are present in the timing model.

value is the minimum period limit and cannot be negative.

Example



```
(INSTANCE x.b)
(TIMINGCHECK
  (PERIOD (posedge clk) (46.5))
```

In this example, the data applies to a minimum period check between consecutive rising edges.

As with all *port_tchk*s, the **COND** construct can be used to specify conditions associated with the minimum period timing check.

No Change Timing Check

The **NOCHANGE** entry specifies limit values for a nochange timing check. The nochange timing check is a signal check relative to the width of a control pulse. A "setup" period is established before the start of the control pulse and a "hold" period after the pulse. The signal checked against the control signal must remain stable during the setup period, the entire width of the pulse and the hold period. A typical use of a nochange timing check is to model the timing of memory devices, when address lines must remain stable during a write pulse with margins both before and after.

Syntax

(NOCHANGE *port_tchk port_tchk rvalue rvalue* **)**

The first *port_tchk* refers to the control port, which is typically a write enable input to a memory or register file device. An edge specification must be included for the control port.

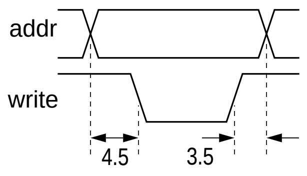
The second *port_tchk* refers to the port checked against the control port, which is typically an address or select input to a memory or register file device. An edge specification can be included.

The first *rvalue* is the minimum time that the data/address must be present (stable) before the specified edge of the control signal (setup).

The second *rvalue* is the minimum time that the data/address must remain stable after the opposite edge of the control signal (hold).

Example

nochange check



```
(INSTANCE x)
(TIMINGCHECK
  (NOCHANGE (negedge write) addr (4.5) (3.5))
)
```

This example defines a period beginning 4.5 time units before the falling edge of `write` and ending 3.5 time units after the subsequent rising edge of `write`. During this time period, the `addr` signal must not change.

As with all *port_tchk*s, the **COND** construct can be used to specify conditions associated with the nochange timing check.

Timing Environment Entries

Timing specifications that start with the **TIMINGENV** keyword associate constraint values with critical paths in the design and provide information about the timing environment in which the circuit will operate. Constructs in this section are used in forward-annotation and not back-annotation.

Syntax

te_spec ::= **(TIMINGENV** *te_def*+ **)**

Any number of *te_def* entries may appear in a *te_spec* entry. Each *te_def* will be a **PATHCONSTRAINT**, **PERIODCONSTRAINT**, **SUM**, **DIFF** or **SKEWCONSTRAINT** constraint entry, containing constraint values for the design or an **ARRIVAL**, **DEPARTURE**, **SLACK** or **WAVEFORM** timing environment entry, containing information about the timing environment in which the circuit will operate.

Syntax

```
te_def  ||=  cns_def            // constraint
        ::=  tenv_def           // timing environment
```

Constraints are covered in the next section. Timing environment is covered in “Constraints” on page 36

Constraints

Constraint entries provide information about the timing properties that a design is required to have in order to meet certain design objectives. A tool that is synthesizing some aspect of the design (logic synthesis, layout, etc.) will adapt its strategy to try to ensure that the constraints are met and issue warning messages in the event that they cannot be met.

Syntax

```
cns_def ::= ( PATHCONSTRAINT name? port_instance port_instance+
                                                   rvalue rvalue )
        ||= ( PERIODCONSTRAINT port_instance value exception? )
        ||= ( SUM constraint_path constraint_path+ rvalue rvalue? )
        ||= ( DIFF constraint_path constraint_path value value? )
        ||= ( SKEWCONSTRAINT port_spec value )
```

The following sections describe the SDF constraint constructs.

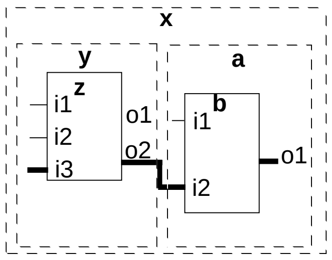
Path Constraint

The **PATHCONSTRAINT** entry represents delay constraints for paths. Path constraints are the critical paths in a design identified during timing analysis. Layout tools can use these constraints to direct the physical design. The constraint specifies the maximum allowable delay for a path, which is typically identified by two ports, one at each end of the path. You can also specify intermediate ports to uniquely identify the path.

Syntax

(PATHCONSTRAINT *name? port_instance port_instance+ rvalue rvalue)*

name ::= **(NAME** QSTRING **)**



name is optional and allows a symbolic name to be associated with the path. This name should be used by the tool to identify the path to the user when information about the path (problems, failures, etc.) is to be provided. The name is assumed to be more convenient for this purpose than the list of port instances.

The first *port_instance* is the start of the path.

The last *port_instance* is the end of the path. You can specify intermediate points along the path by using additional *port_instances* in this entry.

The first *rvalue* is the maximum rise delay between the start and end points of the path.

The second *rvalue* is the maximum fall delay between the start and end points of the path.

Example

```
(INSTANCE x)
(TIMINGENV
  (PATHCONSTRAINT y.z.i3 y.z.o2 a.b.o1 (25.1) (15.6))
)
```

Period Constraint

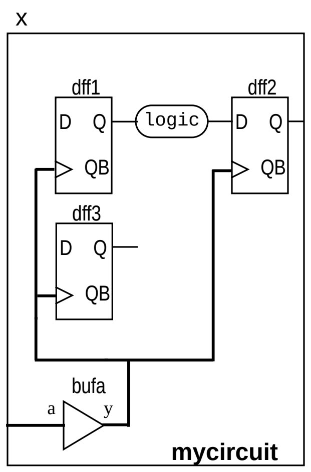
The **PERIODCONSTRAINT** construct allows a path constraint value to be specified for groups of paths in a synchronous circuit. All paths in the group will be from the common clock input of some flip-flops to the data inputs of the flip-flops that share the common clock. This can be used to derive the frequency at which a circuit must operate as a constraint on how long signals can take after a clock edge to reach the register data inputs.

Syntax

(PERIODCONSTRAINT *port_instance value exception*? **)**

exception ::= **(EXCEPTION** *cell_instance+* **)**

port_instance identifies the common clock signal which is the start of all constrained paths. Whereas the start of a **PATHCONSTRAINT** entry is normally an input port, *port_instance* here is normally the output port of the device that drives the clock of the flip-flops. Only flip-flops directly connected to this output are in constrained paths. Paths that pass through other buffers before reaching a flip-flop clock are also considered in the group constrained by this entry.



value is the maximum allowable delay for each path in the group. Included in this delay is the clock-to-output delay of the flip-flop driven from *port_instance,* the setup time of the flip flop that ends the path, and the delay through any combinational logic before arrival at the data input of a flip-flop. Not included is the difference in the timing of the clock of that flip-flop that ends the path from the clock that starts the path. These two times will cause the *value* supplied in a **PERIODCONSTRAINT** entry to be different (typically smaller) than the intended clock period at which the circuit will operate. Since only one *value* can be supplied for all paths in this group, some data may be lost in combining many **PATHCONSTRAINT** entries into one **PERIODCONSTRAINT** entry.

exception is optional and allows paths to be excluded from the group by the identification of a cell through which they pass. One or more cell instances can be listed after the **EXCEPTION** keyword. The hierarchical path to these cell instances is relative to the scope or design region identified by the cell entry. Therefore, the **PERIODCONSTRAINT** entry must appear at a hierarchical level that includes the cell instance that drives the common clock inputs of the flip-flops and any cell instances to be placed in the *exception* list.

Example

```
(INSTANCE x)
(TIMINGENV
  (PERIODCONSTRAINT bufa.y (10)
    (EXCEPTION (INSTANCE dff3) )
  )
)
```

Clearly, any tool that makes use of **PERIODCONSTRAINT** entries in SDF must be able to traverse the design topology and recognize flip-flops and their clock and data inputs.

Sum Constraint

The **SUM** entry represents a constraint on the sum of the delay over two or more paths in a design.

Syntax

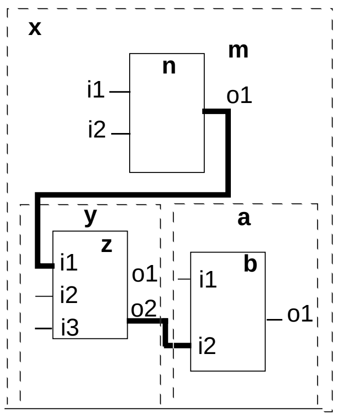
(SUM *constraint_path constraint_path*+ *rvalue rvalue*? **)**

constraint_path ::= **(** *port_instance port_instance* **)**

Each *constraint_path* specifies a path to be included in the sum. You must specify at least two paths, but can specify more.

In each *constraint_path* the first *port_instance* is the beginning of the path and the second *port_instance* is the end of the path.

rvalue is the constraint value. The total (sum) of the individual delays associated with each *constraint_path* must be less than *rvalue*. If two



*rvalue*s are supplied, the first applies to the rising transition at the end of the path and the second to the falling.

Example

```
(INSTANCE x)
(TIMINGENV
    (SUM (m.n.o1 y.z.i1) (y.z.o2 a.b.i2) (67.3))
)
```

This example constrains the sum of the delays along the two nets shown as heavy lines in the diagram to be less than 67.3 time units.

Diff Constraint

The **DIFF** entry represents a constraint on the difference in the delay over two paths in a design.

Syntax

(DIFF *constraint_path constraint_path value value*? **)**

constraint_path specifies a path between two ports. You must specify exactly two paths.

In each *constraint_path* the first *port_instance* is the beginning of the path and the second *port_instance* is the end of the path.

value is the constraint value and must be a positive number or zero. The absolute value of the difference of the individual delays in the two circuit paths must be less than *value*. If two *value*s are supplied, the first applies to the rising transition at the end of the path and the second to the falling.

Example

```
(INSTANCE x)
(TIMINGENV
  (DIFF (m.n.o1 y.z.i1) (y.z.o2 a.b.i2) (8.3) )
)
```

Skew Constraint

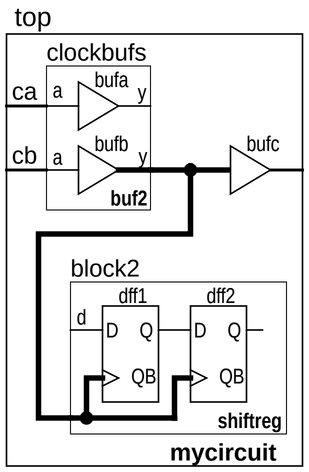
The **SKEWCONSTRAINT** entry represents a constraint on the spread of delays from a common driver to all driven inputs. Only the driving output port can be specified in this construct. All inputs connected to this output are implied end-points for constrained paths. Only paths over interconnect can be constrained as these implied paths cannot pass through any active devices.

Syntax

(SKEWCONSTRAINT *port_spec value* **)**

port_spec refers to the port driving the net.

value is the constraint value and must be a positive number or zero (although zero clock skew might be a hard constraint for a layout tool to meet!). The delays from the output specified by *port_spec* to all inputs that it drives may not differ from each other by more than *value*. This does not place a constraint on the actual value of the delays, just their "spread".



Example

```
(CELL
  (CELLTYPE "buf")
  (INSTANCE top.clockbufs)
  (TIMINGENV
    (SKEWCONSTRAINT (posedge y) (7.5))
  )
)
```

In this example, a buffer cell of cell type `buf` is used to drive some clock inputs in a circuit. It is buried in the design hierarchy by being instantiated as `bufb` in a user block called `clockbufs`, which in turn is part of the block `top`. In the excerpt from an SDF file, this buffer is identified in a **CELL** entry and its output is specified in a **SKEWCONSTRAINT** entry. The effect is to request that the arrival of the positive edge of the clock should not deviate by more than 7.5 between all the inputs driven by the heavily drawn net in the diagram. Neither the inputs nor the net name need to be specified in the SDF file entry. Note that the driven inputs can be anywhere in the design, irrespective of the hierarchical organization.

Timing Environment

Timing environment entries provide information about the timing environment in which the circuit will operate. This can be used by analysis tools to determine whether or not a design will operate correctly given the back-annotation timing data given elsewhere in the file. It can also be used to compute constraints to be forward-annotated to subsequent stages in the design synthesis process.

Syntax

tenv_def ::= **(ARRIVAL** *port_edge*? *port_instance rvalue rvalue rvalue rvalue* **)**
||= **(DEPARTURE** *port_edge*? *port_instance*
rvalue rvalue rvalue rvalue **)**
||= **(SLACK** *port_instance rvalue rvalue rvalue rvalue* NUMBER? **)**
||= **(WAVEFORM** *port_instance* NUMBER *edge_list* **)**

The following sections describe the SDF timing environment constructs.

Arrival Time

The **ARRIVAL** construct defines the time at which a primary input signal is to be applied during the intended circuit operation. Tools use this information to analyze the circuit for timing behavior and to compute constraints for logic synthesis and layout.

Syntax

(ARRIVAL *port_edge*? *port_instance rvalue rvalue rvalue rvalue* **)**

port_edge identifies a port and signal edge that form the time reference for the arrival time specification. The port must be an input port. The *port_edge* is required if the primary input signal is a fan-out from a sequential element, in which case, *port_edge* is usually referred to an active edge of a clock signal. Otherwise, the *port_edge* can be omitted. All **ARRIVAL** constructs that do not have the *port_edge* refer to the same implicit time reference point. This reference time should be treated as the time 0 of all **WAVEFORM** constructs. Note that, to fully specify a timing environment, a **WAVEFORM** statement is required for each clock signal.

port_instance specifies the port at which the arrival time is to be defined. It must be an input or bidirectional port that is a primary (external) input of the top-level module.

Four *rvalues* carry the arrival-time data in this order: earliest rising, latest rising, earliest falling and latest falling arrival times. All values are relative to the time reference, either by a *port_edge*, or by the implicit reference point. The earliest arrival times must be less than the latest arrival times for the same transition.

Multiple **ARRIVAL** statements can be defined for the same input to represent signal paths of different reference *port_edge*s.

Example

```
(INSTANCE top)
(TIMINGENV
  (ARRIVAL (posedge MCLK) D[15:0] (10) (40) (12) (45) )
)
```

This example specifies that rising transitions at `D[15:0]` are to be applied no sooner than 10 and no later than 40 time units after the rising edge of the reference clock `MCLK`. Falling transitions are to be applied no sooner than 12 and no later than 45 time units after the edge.

Departure Time

The **DEPARTURE** construct defines the time at which a primary output signal is to occur during the intended circuit operation. Tools use this information to analyze the circuit for timing behavior and to compute constraints for logic synthesis and layout.

Syntax

(DEPARTURE *port_edge*? *port_instance rvalue rvalue rvalue rvalue* **)**

port_edge identifies a port and signal edge that form the time reference for the departure time specification. The port must be an input port. The *port_edge* is required if the primary output is a fan-out from a sequential element, in which case, *port_edge* is usually referred to an active edge of a clock signal. Otherwise, the *port_edge* can be omitted. All

DEPARTURE constructs that do not have the *port_edge* refer to the same implicit time reference point. This reference time should be treated as the time 0 of all **WAVEFORM** constructs. Note that, to fully specify a timing environment, a **WAVEFORM** statement is required for each clock signal.

port_instance specifies the port at which the departure time is to be defined. It must be an output or bidirectional port that is a primary (external) output of the top-level module.

Four *rvalues* carry the departure-time data in this order: earliest rising, latest rising, earliest falling and latest falling departure times. All values are relative to the time reference, either by a *port_edge*, or by the implicit reference point. The earliest departure times must be less than the latest departure times for the same transition.

Multiple **DEPARTURE** statements can be defined for the same output to represent signal paths of different reference *port_edge*s.

Example

```
(INSTANCE top)
(TIMINGENV
  (DEPARTURE (posedge SCLK) A[15:0] (8) (20) (12) (34) )
)
```

The example specifies that rising transitions at primary output `A[15:0]` are to occur no sooner than 8 and no later than 20 time units after the rising edge of the reference clock `SCLK`. Falling transitions are to occur no sooner than 12 and no later than 34 time units after the edge.

Slack Time

The **SLACK** construct is used to specify the available slack or margin in a delay path. This is a comparison of the calculated delay over a path to the delay constraints imposed upon that path. Positive slack indicates that the constraints are met with room to spare. Negative slack indicates a failure to construct the circuit according to the constraints. A layout or logic synthesis tool can use slack information to make trade-offs in cell placement and routing or re-synthesis of parts of the circuit. The objective should be to eliminate negative slack and achieve an even distribution of positive slack.

Syntax

(SLACK *port_instance rvalue rvalue rvalue rvalue* NUMBER? **)**

port_instance specifies the input port at which slack/margin information is given in this entry. Paths terminating at this port have at least the indicated slack/margin. It is not possible in this construct to specify individual paths. The values given must be the minimum of all paths that converge to the specified *port_instance*. However, the slack/margin may be given at various places on the same path.

Four *rvalues* carry the slack/margin data. In order, they are the rising setup slack, the falling setup slack, the rising hold slack and the falling hold slack. “Rising” and “falling” indicate the direction of transitions at the specified *port_instance* to which data applies. The setup slack is the additional delay that could be tolerated in all paths ending at this port without causing design constraints to be violated. Similarly, the hold slack is the reduction of the delay that could be tolerated in all these paths. If *rtriple*s are used in these *rvalue*s, then each number belongs to the data set for that position in the triple. Since the prevailing use of these data sets is to carry data for minimum, typical and maximum delays, setup slack *rtriple*s will have the unusual property of decreasing in value from left to right.

NUMBER is optional and, if present, represents the clock period on which the slack/margin values are based. The clock period refers to the one specified by a **WAVEFORM** construct.

Example

```
(CELL
  (CELLTYPE “cpu”)
  (INSTANCE macro.AOI6)
  (TIMINGENV
    (SLACK B (3) (3) (7) (7))
  )
)
```

In this example, the delay of any or all data paths leading to port `macro.AOI6.B` could be increased by 3 time units without violating a setup requirement on a constrained device down the path traversed by this port. This **SLACK** entry indicates that the signal arrives at port `macro.AOI6.B` in time to meet the setup time requirement of a flip-flop down the path with 3 time units to spare. Thus, the signals could be delayed up the data path by an additional 3 time units with no ill consequences. The example also shows that the delay of any or all datapaths leading to port `macro.AOI6.B` could be decreased by 7 time units without violating a hold requirement on a constrained device down the path.

Multiple **SLACK** entries are allowable for the same *port_instance* and are distinct if NUMBER is different.

Waveform Specification

The **WAVEFORM** construct allows the specification of a periodic waveform that will be applied to a circuit during its intended operation. Typically, this will be used to define a clock signal. Tools can use this information in analyzing the circuit for timing behavior and to compute constraints for logic synthesis and layout.

Syntax

(WAVEFORM *port_instance* NUMBER *edge_list* **)**

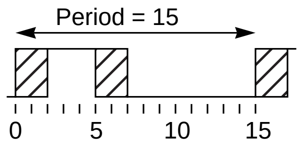
edge_list	::= *pos_pair+* ||= *neg_pair+*
pos_pair	::= **(posedge** RNUMBER RNUMBER? **)** **(negedge** RNUMBER RNUMBER? **)**
neg_pair	::= **(negedge** RNUMBER RNUMBER? **)** **(posedge** RNUMBER RNUMBER? **)**

port_instance identifies the port in the circuit at which the waveform will appear. It must be an input or bidirectional port. If the port is not a primary input of the circuit, i.e. if it is driven by the output of some other circuit element in the scope of the analysis, then the signal driven in the circuit should be ignored and the specified waveform should replace it in the analysis. The hierarchical path to this port is relative to the scope or design region identified by the cell entry.

NUMBER specifies the period of the waveform. The waveform described repeats indefinitely at this interval.

edge_list describes a single period of the waveform. It consists of a list of edge pairs, which can be either a **posedge** entry followed by a **negedge** entry or a **negedge** entry followed by a **posedge** entry. Thus, the total number of edges in the list will be even and edges will alternate between **posedge** and **negedge**. In addition to the direction of the transition, each edge gives the time at which the transition takes place relative to the start of each period. Offsets must increase monotonically throughout the *edge_list* and must not exceed the period. If one RNUMBER is supplied, then this precisely defines the transition offset. If two RNUMBERs are supplied, then they define an uncertainty region in which the transition will take place. The first RNUMBER gives the beginning of the uncertainty region and the second RNUMBER gives its end. Tools using this construct with two RNUMBERs should assume that a single transition of the specified direction occurs somewhere in the uncertainty region, but should make no assumptions about exactly where. Tools unable to model this edge uncertainty should issue a warning message and use the mean of the two RNUMBERs to locate the transition.

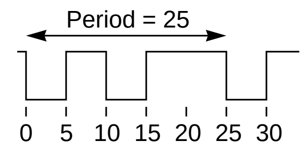
Example



```
(CELL
  (CELLTYPE “cpu”)
  (INSTANCE top)
  (TIMINGENV
    (WAVEFORM clka 15 (posedge 0 2) (negedge 5 7))
  )
)
```

This example shows the specification of a waveform of period 15 to be applied to port `top.clka`. Within each period, a rising edge occurs at somewhere between 0 and 2 and a falling edge somewhere between 5 and 7. Tools unable to deal with uncertainty in waveforms would place the rising edge and 1 and the falling edge at 6 and issue a warning.

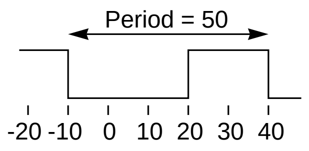
Example



```
(CELL
  (CELLTYPE “cpu”)
  (INSTANCE top)
  (TIMINGENV
    (WAVEFORM clkb 25
      (negedge 0) (posedge 5)
      (negedge 10) (posedge 15)
    )
  )
)
```

This example shows the specification of a waveform of period 25 to be applied to port `top.clkb`. Within each period, a falling edge occurs at 0, a rising edge at 5, a falling edge at 10 and a rising edge at 15.

Example



```
(CELL
  (CELLTYPE “cpu”)
  (INSTANCE top)
  (TIMINGENV
    (WAVEFORM clkb 50
      (negedge -10) (posedge 20)
    )
  )
)
```

This example shows that negative numbers can be used in defining a waveform.

4

Syntax of SDF

SDF File Characters

The legal SDF character set and the method of including comments in SDF files are described in this section.

SDF Characters

The characters you can use in an SDF file are the following:

- Alphanumeric characters – the letters of the alphabet, all the numbers, and the underscore '_' character.
- Special characters – any character other than alphanumeric characters (which includes the underscore as defined above) is a special character. The following is a list of special characters:
  ! " # $ % & ´ ( ) * + , - . / : ; < = > ? @ [\ ] ^ ` { | } ~
- Syntax characters – these are special characters required by the syntax. Examples are: ( ) " * : [ ] ? and the hierarchy divider character but see also the definitions of SDF operators, etc.
- The escape character – to use any special character in an IDENTIFIER, prefix it with the escape character, a backslash '\'. See "Variables" on page 4-2 for a description of an IDENTIFIER. Note that if the character would normally have any special meaning in an IDENTIFIER, this is lost when the character is escaped.
- Hierarchy divider character – either the period '.' or the slash '/' can be established as the hierarchy divider character, see "Hierarchy Divider Entry" on page 3-5. This character only has this special meaning in an IDENTIFIER. An escaped hierarchy divider character loses its meaning as a hierarchy divider.
- White space characters – tabs, spaces and newlines are considered white space. Use white space to separate lexical tokens.

Keywords, IDENTIFIERs, characters, and numbers must be delimited either by syntax characters or by white space.

Comments

Comments can be placed in SDF files using either "C" or "C++" styles.

"C"-style comments begin with /* and end with */. Nesting of "C"-style comments is not permitted. The places in an SDF file where it is legal to put "C"-style comments are not defined by this specification. Different annotators may have different capabilities in this regard.

"C++"-style comments begin with // and continue until the end of the current line (the next newline character). Annotators should ignore the double-slash and any text after them on any line in the file.

Syntax Conventions

Notation

The notation used in presenting the syntax of SDF are as follows:

item	*item* is a symbol for a syntax construct item.
item ::= *definition*	the BNF symbol *item* is defined as *definition.*
item ::= *definition*1 ||= *definition*2	the BNF symbol *item* is defined either as *definition1* or as *definition2.* (any number of alternative syntax definitions may appear)
item?	*item* is optional in the definition (it may appear once or not at all).
*item**	*item* may appear zero or any number of times.
item+	*item* may appear one or more times (but may not be omitted).
KEYWORD	is a keyword and appears in the file as shown. Keywords are shown in uppercase bold for easy identification but are case insensitive.
VARIABLE	is a symbol for a variable. Variable symbols are shown in uppercase for easy identification. Some variables are defined as one of a number of discrete choices (e.g. HCHAR, which is either a period or a slash). Other variables represent user data such as names and numbers.

Variables

This section defines the user data variables used in SDF. Variables which must be one of a number of choices (enumerations) are defined in the main syntax definition which follows.

QSTRING	is a string of any legal SDF characters and spaces, excluding tabs and newlines, enclosed by double-quotes. Except for the double-quote itself, special characters lose their special meaning in a QSTRING.
NUMBER	is a non-negative (zero or positive) real number, for example: 0, 1, 0.0, 3.4, .7, 0.3, 2.4e2, 5.3e-1
RNUMBER	is a positive, zero or negative real number, for example: 0, 1, 0.0, -3.4, .7, -0.3, 2.4e2, -5.3e-1
DNUMBER	is a non-negative integer number, for example: +12, 23, 0
TSVALUE	is a real number followed by a unit. The number must be 1, 10, 100, 1.0, 10.0, or 100.0. The unit must be us, ns, or ps representing microseconds, nanoseconds and picoseconds, respectively. A space may optionally separate the number and unit. Examples of TSVALUE are 1ns, 100 ps, 1us. See also “Timescale Entry” on page 3-7.

IDENTIFIER is the name of an object in the design. This could be an instance of a design block or cell or a port depending on where the IDENTIFIER occurs in the SDF file. Identifiers can be up to 1024 characters long.

The following characters can be used in an identifier:

- Alphanumeric characters – the letters of the alphabet, all the numbers, and the underscore ‘_’ character. IDENTIFIERs are case-sensitive, i.e. uppercase and lowercase letters are considered different.
- Bit specs – to indicate an object selected from an array of objects, for example a single port selected from a bus port or an instance from an array of instances, use a “bit spec” at the end of the IDENTIFIER of the array (with no separating white space). A bit spec consists of square brackets (‘[’ and ‘]’) enclosing a range. To select a single object, the range should be a single positive integer, for example, [4]. To select a contiguous group of objects, the range should be a pair of positive integers separated by a colon (‘:’), for example, [3:31] and [15:0].
- Hierarchy divider character – see “PATH” below.
- The escape character ‘\’ – if you want to use a non-alphanumeric character as a part of an IDENTIFIER it must be escaped by being prefixed with the ‘\’ character. Examples are shown below.
  Note – this escaping mechanism is different from Verilog HDL where the entire IDENTIFIER is escaped by placing one escape character (\) before the IDENTIFIER and a white space after the IDENTIFIER.
  Characters that have special meaning in identifiers, such as ‘[’, ‘]’ and the hierarchy divider, loose that special meaning when escaped.
- Do not use white space (spaces, tabs or newlines) in an IDENTIFIER.

Examples of correct IDENTIFIERs are:

```
AMUX\+BMUX

Cache_Row_\#4

mem_array\[0\:1023\]\(0\:15\)   ; From a language where square
                                ; brackets indicates arrays
                                ; parentheses indicates bit specs

pipe4\-done\&enb[3]             ; Unescaped square brackets
                                ; represent a bit spec
```

PATH is a hierarchical IDENTIFIER. The names of levels in the design hierarchy must be separated by the hierarchy divider character. This character must not be escaped or it looses its meaning as a hierarchy divider. See “Hierarchy Divider Entry” on page 3-5 for details on how the hierarchy divider character is established.

SDF File Syntax

The formal syntax definition for the Standard Delay Format is given here. It is not possible, using the notation chosen, to clearly show how white-space must be used in the SDF file. Some explanations and comments are included in the formal descriptions. A double-slash (//) indicates comments which are not part of the syntax definition.

delay_file	::=	**(DELAYFILE** *sdf_header cell+* **)**
sdf_header	::=	*sdf_version design_name*? *date*? *vendor*? *program_name*? *program_version*? *hierarchy_divider*? *voltage*? *process*? *temperature*? *time_scale*?
sdf_version	::=	**(SDFVERSION** QSTRING **)**
design_name	::=	**(DESIGN** QSTRING **)**
date	::=	**(DATE** QSTRING **)**
vendor	::=	**(VENDOR** QSTRING **)**
program_name	::=	**(PROGRAM** QSTRING **)**
program_version	::=	**(VERSION** QSTRING **)**
hierarchy_divider	::=	**(DIVIDER** HCHAR **)**
HCHAR	::=	**.** // a period character
	||=	/ // a slash character
voltage	::=	**(VOLTAGE** *rtriple* **)**
	||=	**(VOLTAGE** RNUMBER **)**
process	::=	**(PROCESS** QSTRING **)**
temperature	::=	**(TEMPERATURE** *rtriple* **)**
	||=	**(TEMPERATURE** RNUMBER **)**
time_scale	::=	**(TIMESCALE** TSVALUE **)**

Cell Entries

Cell entries are defined as follows:

cell	::=	**(CELL** *celltype cell_instance timing_spec** **)**
celltype	::=	**(CELLTYPE** QSTRING)
cell_instance	::=	**(INSTANCE** PATH? **)**
	||=	**(INSTANCE** WILDCARD **)**
WILDCARD	::=	***** // the asterisk character

Timing Specifications

Timing specifications are defined as follows:

timing_spec	::=	*del_spec*
	||=	*tc_spec*
	||=	*te_spec*
del_spec	::=	**(DELAY** *deltype*+ **)**
tc_spec	::=	**(TIMINGCHECK** *tchk_def*+ **)**
te_spec	::=	**(TIMINGENV** *te_def*+ **)**
deltype	::=	**(PATHPULSE** *input_output_path*? *value value*? **)**
	||=	**(PATHPULSEPERCENT** *input_output_path*? *value value*? **)**
	||=	**(ABSOLUTE** *del_def*+ **)**
	||=	**(INCREMENT** *del_def*+ **)**
input_output_path	::=	*port_instance port_instance*
del_def	::=	**(IOPATH** *port_spec port_instance*
		(RETAIN *delval_list* **)*** *delval_list* **)**
	||=	**(COND** QSTRING? *conditional_port_expr*
		(IOPATH *port_spec port_instance*
		(RETAIN *delval_list* **)*** *delval_list* **))**
	||=	**(CONDELSE**
		(IOPATH *port_spec port_instance*
		(RETAIN *delval_list* **)*** *delval_list* **))**
	||=	**(PORT** *port_instance delval_list* **)**
	||=	**(INTERCONNECT** *port_instance port_instance delval_list* **)**
	||=	**(DEVICE** *port_instance*? *delval_list* **)**
tchk_def	::=	**(SETUP** *port_tchk port_tchk value* **)**
	||=	**(HOLD** *port_tchk port_tchk value* **)**
	||=	**(SETUPHOLD** *port_tchk port_tchk rvalue rvalue* **)**
	||=	**(SETUPHOLD** *port_spec port_spec rvalue rvalue scond*? *ccond*? **)**
	||=	**(RECOVERY** *port_tchk port_tchk value* **)**
	||=	**(REMOVAL** *port_tchk port_tchk value* **)**
	||=	**(RECREM** *port_tchk port_tchk rvalue rvalue* **)**

	||=	**(RECREM** *port_spec port_spec rvalue rvalue scond*? *ccond*? **)**
	||=	**(SKEW** *port_tchk port_tchk rvalue* **)**
	||=	**(WIDTH** *port_tchk value* **)**
	||=	**(PERIOD** *port_tchk value* **)**
	||=	**(NOCHANGE** *port_tchk port_tchk rvalue rvalue* **)**
port_tchk	::=	*port_spec*
	||=	**(COND** QSTRING? *timing_check_condition port_spec* **)**
scond	::=	**(SCOND** QSTRING? *timing_check_condition* **)**
ccond	::=	**(CCOND** QSTRING? *timing_check_condition* **)**
name	::=	**(NAME** QSTRING?**)**
exception	::=	**(EXCEPTION** *cell_instance+* **)**
te_def	::=	*cns_def*
	||=	*tenv_def*
cns_def	::=	**(PATHCONSTRAINT** *name*? *port_instance port_instance+ rvalue rvalue* **)**
	||=	**(PERIODCONSTRAINT** *port_instance value exception*? **)**
	||=	**(SUM** *constraint_path constraint_path+ rvalue rvalue*? **)**
	||=	**(DIFF** *constraint_path constraint_path value value*? **)**
	||=	**(SKEWCONSTRAINT** *port_spec value* **)**
tenv_def	::=	**(ARRIVAL** *port_edge*? *port_instance rvalue rvalue rvalue rvalue* **)**
	||=	**(DEPARTURE** *port_edge*? *port_instance rvalue rvalue rvalue rvalue* **)**
	||=	**(SLACK** *port_instance rvalue rvalue rvalue rvalue* NUMBER? **)**
	||=	**(WAVEFORM** *port_instance* NUMBER *edge_list* **)**
constraint_path	::=	**(** *port_instance port_instance* **)**
port_spec	::=	*port_instance*
	||=	*port_edge*
port_edge	::=	**(** EDGE_IDENTIFIER *port_instance* **)**
EDGE_IDENTIFIER	::=	**posedge**
	||=	**negedge**
	||=	**01**
	||=	**10**
	||=	**0z**
	||=	**z1**
	||=	**1z**
	||=	**z0**
port_instance	::=	*port*
	||=	PATH HCHAR *port*

port	::= *scalar_port*
	||= *bus_port*
scalar_port	::= IDENTIFIER
	||= IDENTIFIER **[** DNUMBER **]**
bus_port	::= IDENTIFIER **[** DNUMBER **:** DNUMBER **]**
edge_list	::= *pos_pair+*
	||= *neg_pair+*
pos_pair	::= **(posedge** RNUMBER RNUMBER? **) (negedge** RNUMBER RNUMBER? **)**
neg_pair	::= **(negedge** RNUMBER RNUMBER? **) (posedge** RNUMBER RNUMBER? **)**

Data Values

Data values in SDF files are defined as follows:

value	::= **(** NUMBER? **)**
	||= **(** *triple*? **)**

A value consists of a NUMBER in parentheses, a *triple* in parentheses or an empty pair of parentheses. Empty parentheses indicate that no value is supplied for a particular data item. This is used primarily where a construct has a list of data items and it is desired to supply a value for an item further down the list but not for earlier items. The empty parentheses mark the places of the earlier items. An annotator should take no action when it encounters empty parentheses. In particular, it should not interpret this in the same way as a value of zero.

triple	::= NUMBER : NUMBER? : NUMBER?
	||= NUMBER? : NUMBER : NUMBER?
	||= NUMBER? : NUMBER? : NUMBER

A *triple* consists of one, two or three colon-separated NUMBERs. Each NUMBER corresponds to a data value in one of three data sets, commonly used (in order) as values under best case/minimum, nominal/typical and worst case/maximum operating conditions. If a NUMBER is omitted, then a value is not included for that data set. At least one NUMBER is required. Both colons must always be present.

rvalue	::= **(** RNUMBER? **)**
	||= **(** *rtriple*? **)**
rtriple	::= RNUMBER : RNUMBER? : RNUMBER?
	||= RNUMBER? : RNUMBER : RNUMBER?
	||= RNUMBER? : RNUMBER? : RNUMBER

Apart from allowing negative numbers (RNUMBER instead of NUMBER), *rvalue* and *rtriple* are essentially the same as *value* and *triple*.

delval ::= *rvalue*
||= **(** *rvalue rvalue* **)**
||= **(** *rvalue rvalue rvalue* **)**

For specifying delay values, *delval* extends *rvalue* by allowing two or three *rvalue*s to be grouped in a further set of parentheses. When this is used, the first *rvalue* specifies the delay, as if a single *rvalue* were given. The second specifies the pulse rejection limit, or “r-limit”, associated with this delay. The third specifies the X-limit, or “e-limit”. This allows pulse control data to be associated in a uniform way with all types of delays in SDF, i.e. **IOPATH**, **PORT**, **INTERCONNECT** and **DEVICE** delays. Note that since any *rvalue* can be an empty pair of parentheses, each type of delay data can be annotated or omitted as the need arises.

delval_list ::= *delval*
||= *delval delval*
||= *delval delval delval*
||= *delval delval delval delval delval*? *delval*?
||= *delval delval delval delval delval delval*
delval delval? *delval*? *delval*? *delval*? *delval*?

The meaning of *delval*s in an *delval_list* is different for lists of length one, two, three, six or twelve. Lists of length four or five are interpreted in the same way as lists of length six with trailing empty parentheses. Simularly, lists of length seven to eleven are interpreted in the same way as lists of length twelve with trailing empty parentheses. A complete discussion of the use of *delval_list* is included in “Specifying Delay Values” on page 3-16.

Conditions for Path Delays

Path delay conditional expressions are used in conjunction with **IOPATH** entries and are defined as follows:

conditional_port_expr	::=	*simple_expression*	
	||=	**(** *conditional_port_expr* **)**	
	||=	UNARY_OPERATOR **(** *conditional_port_expr* **)**	
	||=	*conditional_port_expr* BINARY_OPERATOR *conditional_port_expr*	
simple_expression	::=	**(** *simple_expression* **)**	
	||=	UNARY_OPERATOR **(** *simple_expression* **)**	
	||=	*port*	
	||=	UNARY_OPERATOR *port*	
	||=	SCALAR_CONSTANT	
	||=	UNARY_OPERATOR SCALAR_CONSTANT	
	||=	*simple_expression* QM *simple_expression* CLN *simple_expression*	
	||=	**{** *simple_expression concat_expression*? **}**	
	||=	**{** *simple_expression* **{** *simple_expression concat_expression*? **} }**	
concat_expression	::=	, *simple_expression*	
QM	::=	**?**	// a literal question mark
CLN	::=	**:**	// a literal colon

Conditions for Timing Checks

Timing check conditional expressions are defined as follows:

timing_check_condition	::=	*scalar_node*
	||=	INVERSION_OPERATOR *scalar_node*
	||=	*scalar_node* EQUALITY_OPERATOR SCALAR_CONSTANT
scalar_node	::=	*scalar_port*
		scalar_net
scalar_net	::=	IDENTIFIER

Constants for Expressions

This section defines the logical constants used in SDF conditional port expressions and timing check conditions.

SCALAR_CONSTANT	::=	**1‘b0**	// logical zero
	||=	**1‘b1**	// logical one
	||=	**1‘B0**	// logical zero
	||=	**1‘B1**	// logical one
	||=	**‘b0**	// logical zero
	||=	**‘b1**	// logical one
	||=	**‘B0**	// logical zero
	||=	**‘B1**	// logical one
	||=	**0**	// logical zero
	||=	**1**	// logical one

Operators for Expressions

This section defines the operators used in SDF conditional port expressions and timing check conditions.

```
  UNARY_OPERATOR     ::= +              // arithmetic identity
                     ||= -              // arithmetic negation
                     ||= !              // logical negation
                     ||= ~              // bit-wise unary negation
                     ||= &              // reduction unary AND
                     ||= ~&             // reduction unary NAND
                     ||= |              // reduction unary OR
                     ||= ~|             // reduction unary NOR
                     ||= ^              // reduction unary XOR
                     ||= ^~             // reduction unary XNOR
                     ||= ~^             // reduction unary XNOR (alternative)

INVERSION_OPERATOR   ::= !              // logical negation
                     ||= ~              // bit-wise unary negation

  BINARY_OPERATOR    ::= +              // arithmetic sum
                     ||= -              // arithmetic difference
                     ||= *              // arithmetic product
                     ||=  /             // arithmetic quotient
                     ||= %              // modulus
                     ||= ==             // logical equality
                     ||= !=             // logical inequality
                     ||= ===            // case equality
                     ||= !==            // case inequality
                     ||= &&             // logical AND
                     ||= ||             // logical OR
                     ||= <              // relational
                     ||= <=             // relational
                     ||= >              // relational
                     ||= >=             // relational
                     ||= &              // bit-wise binary AND
                     ||= |              // bit-wise binary inclusive OR
                     ||= ^              // bit-wise binary exclusive OR
                     ||= ^~             // bit-wise binary equivalence
                     ||= ~^             // bit-wise binary equivalence (alternative)
                     ||= >>             // right shift
                     ||= <<             // left shift

EQUALITY_OPERATOR    ::= ==             // logical equality
                     ||= !=             // logical inequality
                     ||= ===            // case equality
                     ||= !==            // case inequality
```

Operation of SDF Equality Operators

This section describes the operation of the equality operators used in SDF conditional port expressions and timing check conditions. These operators return a logical value representing the result of the comparison, which is 1 for TRUE and 0 for FALSE but may also be X.

`a == b` (logical equality) will be TRUE (1) only if `a` and `b` are of known logical state (0 or 1) and equal and FALSE (0) only if `a` and `b` are known and not equal. If either `a` or `b` is X or Z, then the result will be X.

`a` **!=** `b` (logical inequality) will be TRUE (1) only if `a` and `b` are known and not equal and FALSE (0) only if `a` and `b` are known and equal. If either `a` or `b` is X or Z, then the result will be X.

`a === b` (case equality) will be TRUE (1) if `a` and `b` are of the exact same logical state, including the X and Z states, and FALSE (0) otherwise.

`a` **!==** `b` (case inequality) will be TRUE (1) if `a` and `b` are of different logical states, including the X and Z states, and FALSE (0) otherwise.

Precedence Rules of SDF Operators

This section defines the precedence rules of SDF operators in descending order.

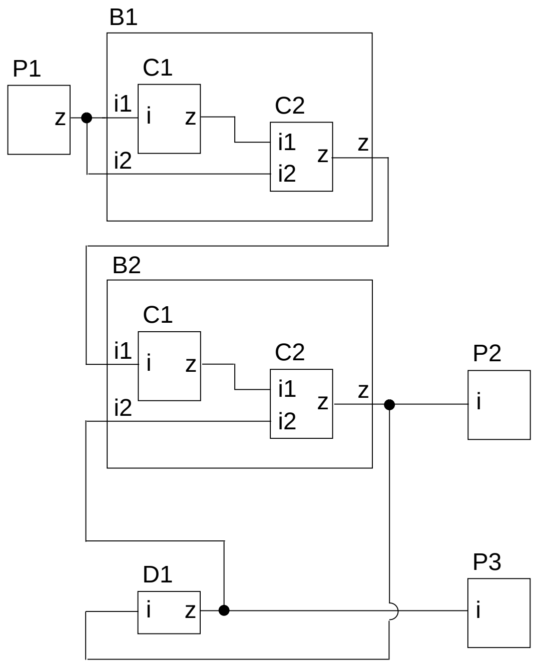
! ~	highest precedence
***** / **%**	
+ **-**	
<< >>	
< <= > >=	
== **!=** === **!==**	
&	
^ ^~	
|	
&&	
||	lowest precedence

SDF File Examples

SDF File Example 1

The SDF file example, shown on the next page, is based on the schematic shown below.

Figure 4 SDF Example Schematic



```
(DELAYFILE
  (SDFVERSION "1.0")
  (DESIGN "system")
  (DATE "Saturday September 30 08:30:33 PST 1990")
  (VENDOR "Yosemite Semiconductor")
  (PROGRAM "delay_calc")
  (VERSION "1.5")
  (DIVIDER /)
  (VOLTAGE 5.5:5.0:4.5)
  (PROCESS "worst")
  (TEMPERATURE 55:85:125)
  (TIMESCALE 1ns)
  (CELL
    (CELLTYPE "system")
    (INSTANCE )
    (DELAY
      (ABSOLUTE
        (INTERCONNECT P1/z    B1/C1/i  (.145::.145) (.125::.125))
        (INTERCONNECT P1/z    B1/C2/i2 (.135::.135) (.130::.130))
        (INTERCONNECT B1/C1/z B1/C2/i1 (.095::.095) (.095::.095))
        (INTERCONNECT B1/C2/z B2/C1/i  (.145::.145) (.125::.125))
        (INTERCONNECT B2/C1/z B2/C2/i1 (.075::.075) (.075::.075))
        (INTERCONNECT B2/C2/z P2/i     (.055::.055) (.075::.075))
        (INTERCONNECT B2/C2/z D1/i     (.255::.255) (.275::.275))
        (INTERCONNECT D1/z    B2/C2/i2 (.155::.155) (.175::.175))
        (INTERCONNECT D1/z    P3/i     (.155::.155) (.130::.130))
      )
    )
  )
  (CELL
    (CELLTYPE "INV")
    (INSTANCE B1/C1)
    (DELAY
      (ABSOLUTE
        (IOPATH i  z (.345::.345) (.325::.325) )
      )
    )
  )
  (CELL
    (CELLTYPE "OR2")
    (INSTANCE B1/C2)
    (DELAY
      (ABSOLUTE
        (IOPATH i1  z (.300::.300) (.325::.325) )
        (IOPATH i2  z (.300::.300) (.325::.325) )
      )
    )
  )
  (CELL
    (CELLTYPE "INV")
    (INSTANCE B2/C1)
    (DELAY
      (ABSOLUTE
        (IOPATH i  z (.345::.345) (.325::.325) )
      )
    )
  )
  (CELL
    (CELLTYPE "AND2")
    (INSTANCE B2/C2)
    (DELAY
      (ABSOLUTE
        (IOPATH i1  z (.300::.300) (.325::.325) )
        (IOPATH i2  z (.300::.300) (.325::.325) )
      )
    )
  )
  (CELL
    (CELLTYPE "INV")
    (INSTANCE D1)
    (DELAY
      (ABSOLUTE
        (IOPATH i  z (.380::.380) (.380::.380) )
      )
    )
  )
)
```

SDF File Example 2

This example shows how you can use the **COND** construct with the **IOPATH** and **TIMINGCHECK** constructs.

```
(DELAYFILE
  (SDFVERSION "2.0")
  (DESIGN "top")
  (DATE "Feb 21, 1992  11:30:10")
  (VENDOR "Cool New Tools")
  (PROGRAM "Delay Obfuscator")
  (VERSION "v1.0")
  (DIVIDER .)
  (VOLTAGE :5:)
  (PROCESS "typical")
  (TEMPERATURE :25:)
  (TIMESCALE 1ns)
  (CELL
    (CELLTYPE "CDS_GEN_FD_P_SD_RB_SB_NO")
    (INSTANCE top.ff1)
    (DELAY
      (ABSOLUTE
        (COND (TE == 0 && RB == 1 && SB == 1)
          (IOPATH (posedge CP) Q (2:2:2) (3:3:3) )
        )
      )
      (ABSOLUTE
        (COND (TE == 0 && RB == 1 && SB == 1)
          (IOPATH (posedge CP) QN (4:4:4) (5:5:5) )
        )
      )
      (ABSOLUTE
        (COND (TE == 1 && RB == 1 && SB == 1)
          (IOPATH (posedge CP) Q (6:6:6) (7:7:7) )
        )
      )
      (ABSOLUTE
        (COND (TE == 1 && RB == 1 && SB == 1)
          (IOPATH (posedge CP) QN (8:8:8) (9:9:9) )
        )
      )
      (ABSOLUTE
        (IOPATH (negedge RB) Q (1:1:1) (1:1:1) ) )
      (ABSOLUTE
        (IOPATH (negedge RB) QN (1:1:1) (1:1:1) ) )
      (ABSOLUTE
        (IOPATH (negedge SB) Q (1:1:1) (1:1:1) ) )
      (ABSOLUTE
        (IOPATH (negedge SB) QN (1:1:1) (1:1:1) ) )
      )
    (DELAY
      (ABSOLUTE
        (PORT D (0:0:0) (0:0:0) (5:5:5) ) )
      (ABSOLUTE
        (PORT CP (0:0:0) (0:0:0) (0:0:0) ) )
      (ABSOLUTE
        (PORT RB (0:0:0) (0:0:0) (0:0:0) ) )
      (ABSOLUTE
        (PORT SB (0:0:0) (0:0:0) (0:0:0) ) )
      (ABSOLUTE
        (PORT TI (0:0:0) (0:0:0) (0:0:0) ) )
      (ABSOLUTE
        (PORT TE (0:0:0) (0:0:0) (0:0:0) ) )
    )
    (TIMINGCHECK
      (SETUP D (COND D_ENABLE (posedge CP)) (1:1:1) )
      (HOLD D (COND D_ENABLE (posedge CP)) (1:1:1) )
      (SETUPHOLD TI (COND TI_ENABLE (posedge CP)) (1:1:1) (1:1:1))
      (WIDTH (COND ENABLE (posedge CP)) (1:1:1) )
      (WIDTH (COND ENABLE (negedge CP)) (1:1:1) )
      (WIDTH (negedge SB) (1:1:1) )
      (WIDTH (negedge RB) (1:1:1) )
      (RECOVERY (posedge RB) (COND SB (negedge CP)) (1:1:1) )
      (RECOVERY (posedge SB) (COND RB (negedge CP)) (1:1:1) )
    )
  )
)
```

SDF File Example 3

This example shows how State Dependent Path Delays can be annotated using **COND** and **IOPATH** constructs.

```
(DELAYFILE
  (SDFVERSION "2.0")
  (DESIGN "top")
  (DATE "Nov 25, 1991 17:25:18")
  (VENDOR "Slick Trick Systems")
  (PROGRAM "Viability Tester")
  (VERSION "v3.0")
  (DIVIDER .)
  (VOLTAGE :5:)
  (PROCESS "typical")
  (TEMPERATURE :25:)
  (TIMESCALE 1ns)
  (CELL
    (CELLTYPE "XOR2")
    (INSTANCE top.x1)
    (DELAY
      (INCREMENT
        (COND i1 (IOPATH i2 o1 (2:2:2) (2:2:2) ) )
      )
      (INCREMENT
        (COND i2 (IOPATH i1 o1 (2:2:2) (2:2:2) ) )
      )
      (INCREMENT
        (COND ~i1 (IOPATH i2 o1 (3:3:3) (3:3:3) ) )
      )
      (INCREMENT
        (COND ~i2 (IOPATH i1 o1 (3:3:3) (3:3:3) ) )
      )
    )
  )
)
```

SDF File Example 4

This example shows how to forward annotate timing constraints. The key to specifying SDF constraints is to identify INSTANCE-PINS of library cells. In the example shown below I2 is an instance and H01 is a PIN (port) on that instance.

```
(DELAYFILE
  (SDFVERSION "3.0")
  (DESIGN "testchip")
  (DATE "Dec 17, 1991 14:49:48")
  (VENDOR "Big Chips Inc.")
  (PROGRAM "Chip Analyzer")
  (VERSION "1.3b")
  (DIVIDER .)
  (VOLTAGE :3.8: )
  (PROCESS "worst")
  (TEMPERATURE : 37:)
  (TIMESCALE 10ps)
  (CELL
    (CELLTYPE "XOR")
    (INSTANCE )
    (TIMINGENV
      (PATHCONSTRAINT I2.H01 I1.N01 (989:1269:1269) (989:1269:1269)
)
      (PATHCONSTRAINT I2.H01 I3.N01 (904:1087:1087) (904:1087:1087)
)
    )
  )
)
```

Delay Model Recommendation

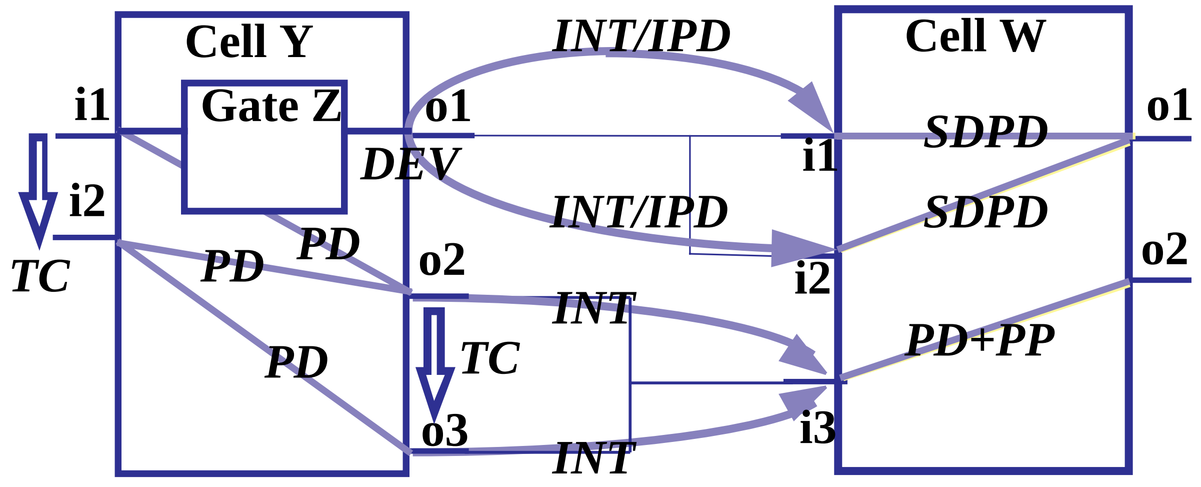
Introduction

The delay model provides a guideline for using SDF in ASIC application tools. All constructs in SDF should be directly applicable to the delay model. ASIC timing is divided into forward annotation and back-annotation. Although SDF supports both timing concepts, this section concentrates on ASIC timing back-annotation model. A future release of SDF will provide an abstract model for forward annotation.

The following section defines the delay model and provides rules that should be adhered to to ensure proper interpretation and usage of SDF constructs.

The Delay Model

Figure 5 The Delay Model



Timing Objects

The delay model consists the following timing objects:

1. Interconnect delay (INT), represented by the **INTERCONNECT** delay construct in SDF.
2. Path delay (PD), represented by **IOPATH** delay construct in SDF.
3. State-dependent path delay (SDPD), represented by **COND** keyword in SDF.
4. Port delay (IPD), represented by **PORT** delay construct in SDF.
5. Device delay (DEV), represented by **DEVICE** construct in SDF. Note when specified with a cell output port, this timing object is a degenerate path delay; when specified with a primitive instance, this timing object is its intrinsic delay.
6. Path pulse (PP), represented by **PATHPULSE** construct in SDF.
7. Timing checks (TC), represented with several keywords in SDF depending on the type of the timing checks.

Rules

1. Path delay is described between any input (or bidirectional) port to any output (or bidirectional) port in the same cell.
2. Multiple path delays can be defined for any output (or bidirectional) port.
3. Multiple path delays can be defined between any pair of ports only by using state dependent delays.
4. Path delay can have up to twelve transition states with twelve different delay values.
5. Negative timing values for absolute input-output path, port, net, device and interconnect delays may default to zero in certain application tools.
6. Interconnect delay is described between any output (bidirectional) port of a cell to any input (bidirectional) port of any cell.
7. Multiple interconnect delays from different sources can be described for any input (bidirectional) port, destination port.
8. Depending on the type of the timing check, it can be applied to a single or a pair of ports.
9. Timing checks are allowed from an output port to another output port.
10. Timing checks are applied after the interconnect delays are applied.
11. Negative timing check limit values are allowed only for the SDF **SETUPHOLD**, **RECREM** and **NOCHANGE** constructs. Some application tools may use the negative values while others may compile them as zero values.
12. **INTERCONNECT** delay between a source and a destination signal cannot be used if **PORT** delay is specified for the same destination signal.
13. Similarly, **PORT** delay for a destination signal cannot be used if an **INTERCONNECT** delay is specified between a source and the same destination signal.
14. **IOPATH** delay cannot be used if a **DEVICE** delay is specified for the same output port within the same cell.
15. Similarly, **DEVICE** delay cannot be used if an **IOPATH** delay is specified between an input port and the same output port within the same cell.
16. All timing objects using the internal nodes may be ignored by application tools that have no concept of the internal nodes.

17. For the same timing object, delay annotation is executed in the sequential order as encountered in a single SDF file.