IEEE Standards

IEEE Std 1497-2001

1497™

IEEE Standard for Standard Delay Format (SDF) for the Electronic Design Process

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

IEEE

Published by
The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

14 December 2001

Print: SH94966
PDF: SS94966

IEEE Std 1497-2001

IEEE Standard for Standard Delay Format (SDF) for the Electronic Design Process

Sponsor

Design Automation Standards Committee of the IEEE Computer Society

Approved 5 December 2001

IEEE-SA Standards Board

Abstract: The Standard Delay Format (SDF) is defined in this standard. SDF is a textual file format for representing the delay and timing information of electronic systems. While both human and machine readable, in its most common usage it will be machine written and machine read in support of timing analysis and verification tools, and of other tools requiring delay and timing information. The primary audience for this standard is the implementors of tools supporting the format, but anyone with a need to understand the format's contents will find it useful.

Keywords: computer, computer languages, delay, delay backannotation, digital systems, electronic systems, hardware, hardware design, SDF, timing, timing analysis, timing backannotation, timing verification

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

Print: ISBN 0-7381-3074-5 SH94966
PDF: ISBN 0-7381-3075-3 SS94966

Introduction

(This introduction is not part of IEEE Std 1497-2001, IEEE Standard for Standard Delay Format (SDF) for the Electronic Design Process.)

The Standard Delay Format (SDF) was designed to serve as a simple textual medium for communicating timing information and constraints between EDA tools. The original version was designed by Rajit C. Chandra in 1990 while at Cadence Design Systems, and was intended as a means of communicating macrocell and interconnect delays from Gate Ensemble to Verilog-XL, Veritime and other stand-alone tools requiring timing data.

Because it was originally targeted for annotation to tools using the Verilog language, many SDF constructs are analogous to those in Verilog specify blocks. Those already familiar with the Verilog specify block will find many of the SDF constructs familiar, such as SETUP and PATHPULSE. SDF also includes constructs for annotating interconnect delays, and can be used for forward annotation by specifying path delay constraints from timing analysis to floorplanners, and synthesis and layout tools.

SDF was first introduced into the EDA marketplace in 1991 where it won quick acceptance. Cadence placed SDF in the public domain in 1992 when it turned control over to Open Verilog International (OVI), and OVI delivered the first SDF standard, version 2.0, in June, 1993 (SDF version 1.0 was used by Cadence). OVI has since introduced version 2.1 in February, 1994, and version 3.0 in May, 1995. VHDL (IEEE 1076) also takes advantage of SDF through the VITAL standard.

In 1996 the OVI Board of Directors began an effort to establish SDF as an IEEE standard. With the approval of the IEEE Design Automation Standards Committee (DASC), the OVI Logic Modeling Technical Subcommittee became the IEEE SDF Study Group. With the approval of the Project Authorization Request (PAR) by the IEEE Standards Board on February 10, 1997, this group became the IEEE SDF Working Group.

This IEEE SDF standard builds upon OVI SDF version 3.0, and will be known as version 4.0. The changes from OVI 3.0 to IEEE 4.0 are small (LABEL construct added, NETDELAY construct restored), but the change from OVI standard to IEEE standard is significant, and so this is recognized by a new version number.

Objective

The starting point for the IEEE P1497 SDF Working Group was the OVI LRM version 3.0 SDF standard, with the goal of soliciting further enhancements and improving the quality and rigor of the LRM. Since SDF is already in widespread use, no modifications that would invalidate current usage were considered.

Acknowledgments

This standard is based on work originally developed by Cadence Design Systems, Inc. (in SDF 1.0) and Open Verilog International (in SDF 2.0, 2.1 and 3.0). The IEEE is grateful to Cadence Design Systems and Open Verilog International for permission to use their materials as the basis for this standard.

The IEEE P1497 SDF Working Group organization

Many individuals from many different organizations participated directly or indirectly in the standardization process. All members of the IEEE P1497 SDF Working Group are located in the United States and had voting privileges. All motions had to be approved by this group to be implemented.

At the time this standard was approved, the IEEE P1497 SDF Working Group had the following membership:

The IEEE P1497 core working group gratefully acknowledges the invaluable contributions of the following individuals, either on the OVI Logic Modeling Technical Subcommittee, in the early stages of the IEEE P1497 SDF Working Group, or in some other valuable role:

The following members of the balloting committee voted on this standard. Balloters may have voted for approval, disapproval, or abstention.

Contents

IEEE Standard for Standard Delay Format (SDF) for the Electronic Design Process

1. Overview

1.1 Scope

The Standard Delay Format (SDF) is an existing OVI standard for the representation and interpretation of timing data for use at any stage of the electronic design process. The ASCII data in the SDF file is represented in a tool and language independent way and includes path delays, timing constraint values, interconnect delays and high level technology parameters. This standard describes the IEEE version of the SDF standard.

This standard should serve as a complete specification of the Standard Delay Format (SDF). It contains:

— Detailed information on how SDF is used in the design process.
— Detailed semantic descriptions of all SDF constructs.
— The formal syntax.
— Examples.

1.2 Organization of this standard

A synopsis of the clauses and annexes of this standard is presented as a quick reference. There are five clauses and two annexes. All the clauses and annexes are normative parts of this standard, with the exception of Annex B (informative).

Clause 1: Overview—Content overview.

Clause 2: References—References to other applicable standards that are assumed or required for SDF.

Clause 3: Definitions and conventions—Introduction to syntactic style and the major syntactic components.

Clause 4: SDF in the design process—The role and use of SDF in the design process.

Clause 5: Defining the Standard Delay Format—The content of an SDF file. For each part of the file, the purpose is discussed, the syntax is specified, the semantics are explained, and examples are presented.

Annex A: Syntax of SDF—SDF file syntax description. The syntax of the contents of an SDF file is described in this annex.

Annex B: SDF file examples—Informative examples of SDF files.

2. References

This standard shall be used in conjunction with the following publications. When the following standards are superseded by an approved revision, the revision shall apply.

IEEE Std 1076, 2000 Edition, IEEE Standard VHDL Language Reference Manual.[1]

IEEE Std 1364-2001, IEEE Standard Verilog® Hardware Description Language.

3. Conventions

3.1 Terminology conventions

The verb "shall" is used throughout this standard to indicate mandatory requirements, whereas the verb "can" is used to indicate optional features that can be used at discretion. If "can" is used, however, one must follow the requirements set forth by the format definition. The verb "shall" denotes different meanings to different readers of this standard:

a) To the developers of tools that process SDF, the verb "shall" denotes a requirement that the standard imposes. The resulting implementation is required to enforce the requirements and to issue an error if the requirement is not met by the input.
b) To the human reader of SDF, the verb "shall" denotes that those characteristics of SDF are natural consequences of the format definition. The characteristics thereby implied in the SDF source text can be depended upon.
c) To the developer of tools that write SDF, and to the human writer of SDF, the verb "shall" denotes that those characteristics of SDF are natural consequences of the format definition. Adherence to the constraint implied by the characteristic is required.

3.2 Syntactic conventions

3.2.1 Syntactic conventions

The formal syntax of SDF is described using Backus-Naur Form (BNF). In addition, the following conventions are used:

a) Lowercase italic words, some containing embedded underscores, are used to denote syntactic tokens. For example:
   module_declaration
b) Boldface words are used to denote reserved keywords, operators, and punctuation marks as a required part of the syntax. For example:
   IOPATH
   (
)

[1]IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA (http://standards.ieee.org/).

c) A vertical bar separates alternative items unless it appears in boldface, in which case it stands for itself. In most cases each alternative appears on a separate line. For example:

   character ::=
   alphanumeric
   |escaped_character

   When the alternatives are very simple, as in the case of single characters, then they can appear on a single line or on consecutive multiple lines. For example:

   decimal_digit ::= ***0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9***

d) Square brackets enclose optional items. For example:

   real_number ::= integer [**.** *integer]*

e) Braces enclose a repeated item unless it appears in boldface, in which case it stands for itself. The item can appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus, this rules says that a CELL can contain any number of timing specifications:

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

A `constant-width` font is used for examples, file names, and while referring to constants, especially `0`, `1`, `x`, and `z` values.

3.2.2 Lexical tokens

An SDF file is a stream of lexical tokens in free format, each of which consists of one or more characters. Spaces and newlines serve only to separate tokens.

3.2.3 White space

Tabs, spaces, and newlines are considered white space. White space is never significant except when used within quoted strings or to separate lexical tokens.

3.2.4 Comments

Comments can be placed in SDF files using either the “C” or “C++” style.

“C”-style comments begin with /* and end with */. Nesting of “C”-style comments is not permitted. “C”-style comments can appear anywhere except within lexical tokens or quoted strings.

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

3.2.5 Identifiers

Identifiers can consist of alphanumeric characters and special characters. Alphanumeric characters consist of the letters of the alphabet, the numeric base-10 digits, the underscore (‘_’), and the dollar sign (‘$’). Special characters must be escaped (preceded with the backslash (‘\’) character) in order to be used in an identifier. The special characters are:

! " # $ % & « ( ) * + , - . / : ; < = > ? @ [\ ] ^ ‘ { | } ~

Any character can be escaped with a backslash, and the backslash is only required for special characters. Note that if a character normally has any special meaning in an identifier, this is lost when the character is escaped.

3.2.6 Quoted strings

A quoted string is a string of any legal SDF characters, including white space, that are enclosed between double-quotes (‘"’). Except for the double-quote itself, special characters lose their special meaning in a quoted string. The double-quote character may be included in a quoted string by escaping it [preceding it with the backslash (‘\’) character].

3.2.7 Bit specifications

A bit specification is indicated by an identifier with trailing paired square brackets (‘**[**’ and ‘**]**’). A single bit is indicated by a single integer between the square brackets, while a bit range is indicated by two integers separated by a colon (‘**:**’).

3.2.8 Hierarchy divider character

Either the period (‘**.**’) or the slash (‘/’) can be established as the hierarchy divider character, as described in 5.2.7. This character only has this special meaning when used to separate identifiers. An escaped hierarchy divider character loses its meaning as a hierarchy divider.

3.2.9 Data values

A number shall be an integer or a real number. Real numbers can be expressed in scientific notation, and can be signed or unsigned, but signed real numbers are not legal in all contexts.

A value consists of a real_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 shall take no action when it encounters empty parentheses. In particular, it shall not interpret this in the same way as a value of zero.

A *triple* consists of one, two or three colon-separated real_numbers. Each real_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 real_number is omitted, then a value is not included for that data set. At least one real_number is required. Both colons must always be present.

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

For specifying delay values, *delval* extends *rvalue* by allowing two or three *rvalue* constructs 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**, **NETDELAY**, 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.

The meaning of *delval* constructs 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. Similarly, 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 5.4.1.

3.2.10 Operators

Operators are single-, double-, or triple-character sequences and are used in expressions.

The equality operators used in SDF conditional port expressions and timing check conditions return a logical value representing the result of the comparison, which is 1 for TRUE and 0 for FALSE, but can 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 shall 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.

4. SDF in the design process

4.1 Sharing of timing data

By accessing an SDF file, Electronic Design Automation (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 post-layout delay data.

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

4.2 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).

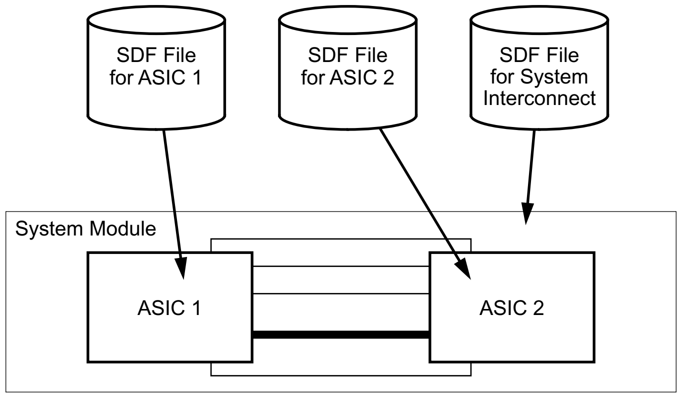


Figure 1—Multiple SDF files in a hierarchical design

4.3 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 floorplanners) may need access to computed timing data as well as the timing constraints intended to be meet.

Subclauses 4.5 and 4.6 discuss the use of SDF for backward- and forward-annotation of timing information.

4.4 Timing environments

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

4.5 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 can have different restrictions in the way in which the data in an SDF file is used. For example, static timing analysis tools may be able to take into account path delays that have a negative value, whereas dynamic timing simulation tools may have to interpret such negative delays as zero. Even though by using SDF the timing data used by each tool is the same, differences in tool capabilities can nevertheless result in small differences in analysis results.

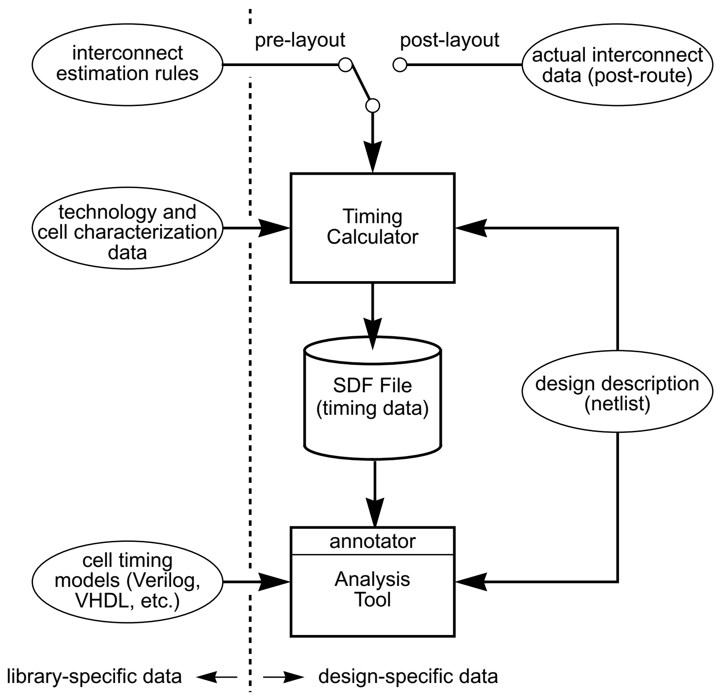

interconnect estimation rules
pre-layout
post-layout
actual interconnect data (post-route)
technology and cell characterization data
Timing Calculator
SDF File (timing data)
design description (netlist)
annotator
Analysis Tool
cell timing models (Verilog, VHDL, etc.)
library-specific data
design-specific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

Figure 2—SDF files in timing back-annotation

4.5.1 The timing calculator

A timing calculator tool is responsible for generating the SDF file. To do this, the timing calculator shall examine the specific design for which it has been instructed to calculate timing data. Figure 2 shows how the timing calculator reads in 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 computation 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 can be obtained by accessing them directly (not shown) or can be built into the timing calculator and/or cell characterization data.

As the timing characteristics of ASICs are strongly influenced by interconnect effects, Figure 2 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 shall 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.

4.5.2 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 shall be applied to the appropriate parameters of the timing model.

The annotator can 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 shall search for regions identified in the SDF file starting at this point in the hierarchy. The file must clearly have been prepared with such usage in mind, otherwise the annotator will be unable to match the data found in the file with the design viewed from this point in hierarchy.

The foregoing implies that the annotator must have access to the design description and the timing models. Frequently, such access is provided via the internal representations maintained by the analysis tool. The annotator then becomes a part of the tool. As an alternative, the annotator can 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 can be unpredictable. Also, certain constructs of SDF cannot be supported without access to the design description (for example, wildcard cell instance specifications).

Definition of all timing relationships, including delays and timing checks shall reside with the timing model. SDF annotation shall not be used to specify timing relationships, but only to communicate timing values.

4.5.3 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 rerun 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 succeed 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 shall be considered errors.

4.5.4 Consistency between SDF file and timing models

An SDF file contains only timing data. It does not contain instructions to the analysis tool concerning how to model the timing properties of the design. The SDF keywords and constructs that 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 be 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 has 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 to distributed delays or a single all-inputs-to-the-output delay).

Some analysis tools and the corresponding 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 cooperate to add this information when the design and timing are loaded or merged in the tool. In this case, the SDF file shall contain data that has no obvious placeholders in the models. Nevertheless, the data must be consistent with the capabilities of the tool to model circuit timing using that data. For example, if interconnect timing is described 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 can 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.

4.6 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 logic synthesis, floorplanner, and layout and routing tools. Timing constraints are the requirements imposed on the overall timing properties of the design, 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.

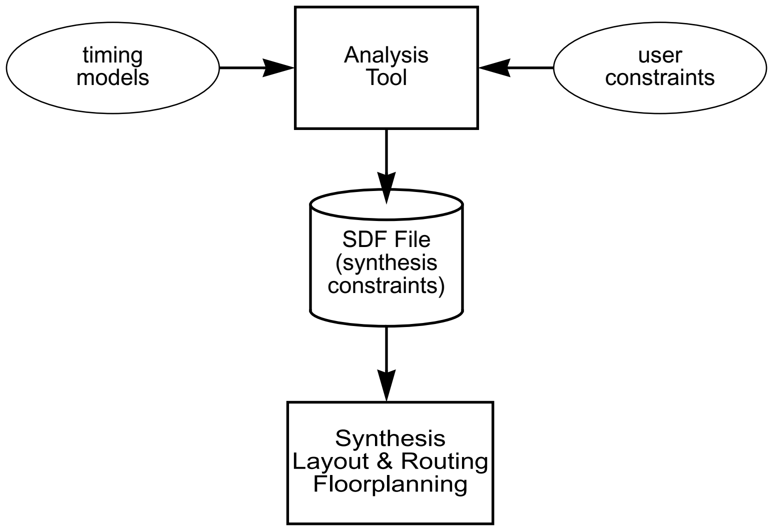

timing models
Analysis Tool
user constraints
SDF File (synthesis constraints)
Synthesis Layout & Routing Floorplanning

Figure 3—SDF files in constraint forward-annotation

For example, the initial requirement might be that the primary clock should run at 50 MHz. 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 floorplanner, and 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 can 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 standard, *timing check* implies a test that an analysis tool performs to make sure that a circuit, as presently constructed, shall operate reliably. The terms *timing constraint* or *constraint* imply a restriction on the timing properties of a design specified to a tool that constructs or modifies some aspect of the design (e.g., logic, layout, or routing).

4.7 Timing models supported by SDF

The importance of the consistency of an SDF file with the timing models has been stressed in 4.5.4. SDF provides a variety of ways to describe the timing of a circuit, allowing considerable flexibility in the design of the timing models. Subclauses 4.7.1 through 4.7.5 describe some modeling methodologies supported by SDF and establishes a consistent terminology used later in describing SDF itself.

4.7.1 Modeling circuit delay

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 **IOPATH** construct shall apply delay data to input-to-output path delays across cells described in this way.
— The **COND** construct shall allows any path delay to be made conditional, that is, the delay 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 applicable circuit state.
— 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 **DEVICE** construct shall apply delay data to modeling primitives in distributed delay models.

4.7.2 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 shall affect the output and a larger limit can be established for the shortest pulse that shall appear with its true logical value, rather than appearing as a "glitch" to the unknown state.

— The **PATHPULSE** construct shall allow these limits to be specified as time values.
— The **PATHPULSEPERCENT** construct shall allow these limits to be specified as percentages of the path delay.

4.7.3 Modeling timing checks

SDF timing check constructs permit values for the following categories of timing checks to be specified:

— setup
— hold
— recovery
— removal
— maximum skew
— minimum pulse width
— minimum period
— no-change

Library models can specify timing checks with respect to both external ports and internal signals. Conditions for ports and signals of timing checks can be specified using the **COND** construct. Negative values are permitted on timing checks where this is meaningful, although analysis tools that cannot use negative values can substitute a value of zero.

4.7.4 Modeling interconnect delays

SDF supports three styles of interconnect delay modeling.

— The **INTERCONNECT** construct shall allow interconnect delays to be specified on a point-to-point source/load basis. This is the most specific method for specifying interconnect delays. It provides the greatest power and flexibility in defining unique source/load delays.
— The **PORT** construct shall allow interconnect delays to be specified to load ports without regard to which source the signal arrives from. This is equivalent to the **INTERCONNECT** construct for wires/nets that have only one driver or source. However, for nets with more than one driver it is not be possible to represent the unique delay from each source.
— The **NETDELAY** construct shall allow the delays to all the load ports of a net to be given the same interconnect delay value. This is the least specific method for specifying interconnect delay, as it is not possible to specify either sources or loads.

4.7.5 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 cannot be portable to tools that do not share the same concept of internal nodes or have models constructed with or without different internal nodes.

5. Defining the standard delay format

Clause 5 describes the content of an SDF file. For each part of the file, the purpose is discussed, the syntax is specified, the semantics are explained, and examples are presented. A complete, formal definition of the file syntax is contained in Annex A.

5.1 SDF file content

SDF files shall be ASCII text files. Every SDF file shall contain a header section followed by one or more cells. The syntax is given in Syntax 1.

```
delay_file ::=
        ( DELAYFILE sdf_header cell { cell } )
```

Syntax 1: Syntax for delay format

The header section, *sdf_header*, shall contain 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 5.2).

Each cell, *cell*, shall identify part of the design (a "region" or "scope") and contain data for delays, timing checks, constraints and the timing environment (see 5.3). A *cell* can 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* can encompass the entire design.

Example (1):

```
(DELAYFILE                   // Header Section
   (SDFVERSION  "4.0")
   (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                     // Cell 1
      (CELLTYPE "BIGCHIP")
      (INSTANCE top)
      (DELAY
         (ABSOLUTE
            (INTERCONNECT mck b/c/clk (.6:.7:.9))
            (INTERCONNECT d[0] b/c/d (.4:.5:.6))
         )
      )
   )

   (CELL                     // Cell 2
      (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                     // Cell 3
      (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                     // Cell4.. Cell n
   . . .
   )
)
```

5.2 Header section

The header section of an SDF file shall contain information that relates to the file as a whole. Except for the SDF version, other items in the header sections shall be optional. The header section syntax defines a strict order for header items and those that are present shall follow this order. The items in the header section are also referred to as *entries* or *fields*.

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 dividers and time scales shall, if present, change the way in which the file is interpreted. The SDF header syntax is given in Syntax 2.

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

Syntax 2: Syntax for the SDF header section

5.2.1 SDF version

The SDF version shall identify the version of the Standard Delay Format specification to which the file conforms. Syntax 3 gives the syntax for SDF version.

sdf_version ::=
    (SDFVERSION *qstring* **)**

Syntax 3: Syntax for the SDF version

qstring shall be a character string, in double quotes. The first substring within *qstring* that matches one of the strings "1.0," "2.0," "2.1," "3.0," or "4.0" shall identify the SDF version. A matching substring shall be present. Other characters before and after this substring shall be permitted and shall be ignored by readers when determining the SDF version.

Example (2):

```
(SDFVERSION "IEEE 1497 4.0")
```

Version strings "1.0," "2.0," "2.1," and "3.0" are versions of the OVI SDF standard. This standard is backward compatible with the predecessor OVI version 3.0. The **SDFVERSION** statement is required.

5.2.2 Design name

The design name shall be an optional field that specifies the name of the design to which the timing data in the file shall apply. It shall be for documentation purposes and shall not affect the meaning of the data in the file. The design name has the form given in Syntax 4.

design_name ::=
(**DESIGN** *qstring* **)**

Syntax 4: Syntax for design name

qstring shall be a name that identifies the design.

5.2.3 Date

The date shall be an optional field that specifies the currency of the data in the file. It shall be for documentation purposes and shall not affect the meaning of the data in the file. The syntax for date is given in Syntax 5.

date ::=
(**DATE** *qstring* **)**

Syntax 5: Syntax for the date

qstring can be any legal string, but it is intended to be a character string that represents the date and/or time when the data in the SDF file was generated.

Example (3):

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

5.2.4 Vendor

The vendor field shall be an optional field that specifies 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 shall be for documentation purposes and shall not affect the meaning of the data in the file. The syntax for vendor field is described in Syntax 6.

vendor ::=
(**VENDOR** *qstring* **)**

Syntax 6: Syntax for the vendor

qstring can be any legal string, but it is intended to be a character string containing the name of the vendor whose tools generated the SDF file.

Example (4):

```
(VENDOR “Acme Semiconductor”)
```

5.2.5 Program name

The program name shall be an optional field that specifies the name of the program that created the file. It shall be for documentation purposes and shall not affect the meaning of the data in the file. The program name has the syntax shown in Syntax 7.

program_name ::=
(PROGRAM *qstring***)**

Syntax 7: Syntax for program name

qstring can be any legal string, but it is intended to be a character string containing the name of the program used to generate the SDF file.

Example (5):

```
(PROGRAM “timcalc”)
```

5.2.6 Program version

The program version shall be an optional field that specifies the version of the program that created the file. It shall be for documentation purposes and shall not affect the meaning of the data in the file. The syntax for program version is given in Syntax 8.

program_version ::=
(VERSION *qstring* **)**

Syntax 8: Syntax for program version

qstring can be any legal string, but it is intended to be a character string containing the program version number used to generate the SDF file.

Example (6):

```
(VERSION “version 1.3”)
```

5.2.7 Hierarchy divider

The hierarchy divider shall be an optional field that specifies which of the two permissible characters shall be used in the file to separate elements of a hierarchical path. The hierarchy divider has the form given in Syntax 9.

hierarchy_divider ::=
(DIVIDER *hchar* **)**

Syntax 9: Syntax for hierarchy divider

hchar shall be either a period (‘**.**’), or a slash (‘/’). It shall not be in quotes.

Example (7):

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

In Example (7), the hierarchy divider is specified to be the slash ('/') character and hierarchical paths use slash rather than period ('**.**') to separate elements.

If the SDF file does not contain a hierarchy divider then the default hierarchy divider shall be the period ('**.**'). See also the descriptions of *identifier* and *hierarchical_identifier* in A.1.

5.2.8 Voltage

The voltage shall be an optional field that specifies the operating voltage or voltages for which the data in the file was computed. It shall be for documentation purposes and shall not affect the meaning of the data in the SDF file. The syntax for voltage is given in Syntax 10.

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

Syntax 10: Syntax for the voltage

rtriple or *signed_real_number* shall indicate the operating voltage (in volts) at which the design timing was calculated or the constraints are to apply.

Example (8):

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

Although this information cannot used by the annotator, it shall 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 specified first in the voltage and the lowest voltage is specified last.

5.2.9 Process

The process shall be an optional field that specifies the process factor for which the data in the file was computed. It shall be for documentation purposes and shall not affect the meaning of the data in the file. The syntax for process is described in Syntax 11.

process ::=
  (PROCESS *qstring* **)**

Syntax 11: Syntax for process

qstring shall be a character string which specifies the process operating envelope.

Example (9):

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

5.2.10 Temperature

The temperature shall be an optional field that specifies the operating temperature for which the data in the file was computed. It shall be for documentation purposes and shall not affect the meaning of the data in the file. The syntax for temperature is given in Syntax 12.

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

Syntax 12: Syntax for temperature

rtriple or *signed_real_number* shall indicate the operating ambient temperature(s) of the design in degrees Celsius (centigrade).

Example (10):

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

5.2.11 Timescale

The timescale shall be an optional field that specifies the units used for all time values in the SDF file. The syntax for time scale is shown in Syntax 13.

time_scale ::=
    (TIMESCALE *timescale_number timescale_unit* **)**
timescale_number ::= **1** | **10** | **100** | **1.0** | **10.0** | **100.0**
timescale_unit ::= **s** | **ms** | **us** | **ns** | **ps** | **fs**

Syntax 13: Syntax for the timescale

TIMESCALE accepts a number followed by a unit. The number can be 1, 10, 100, 1.0, 10.0, or 100.0. The unit can be s, ms, us, ns, ps, or fs, representing seconds, milliseconds, microseconds, nanoseconds, picoseconds, and femtoseconds, respectively. One or more space characters can optionally separate the number and the unit.

Example (11):

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

Example (11) indicates that all time values in the file are to be multiplied by 100 picoseconds. Thus, the values supplied in the **IOPATH** are (0.2ns:0.3ns:0.4ns) and (0.5ns:0.6ns:0.7ns).

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

5.3 Cells

A cell shall identify a particular “region” or “scope” within a design and shall contain timing data to be applied there. For example, a cell may identify a 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. The cells can also identify all occurrences of a particular ASIC library physical primitive, such as a certain type of gate or flip-flop. Data shall be applied to all such library-specific regions in the design. The syntax for a cell is given in Syntax 14.

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

Syntax 14: Syntax for cell

The *celltype* and *cell_instance* fields shall identify a region of the design. The *timing_spec* field shall contain the timing data.

Example (12):

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

An SDF file shall contain any number of cells (other than zero). The order of the cells shall be significant only if they have overlapping effect, in other words, if data from two different cells applies to the same timing property in the design. In this situation, the cells shall be processed strictly from the beginning of the file towards the end and the data they contain shall be applied in sequence to whatever region is appropriate to that cell. If data is applied to a timing property previously referenced by an SDF file, the new data shall be applied over the old and the final value shall be the cumulative effect according to whether the data is applied as a replacement for the old value (**ABSOLUTE** and **TIMINGCHECK** sections) or is added to it (**INCREMENTAL** section).

5.3.1 Cell type

The **CELLTYPE** shall indicate the name of the cell. The syntax for cell type is described in Syntax 15.

celltype ::=
    (CELLTYPE *qstring* **)**

Syntax 15: Syntax for cell type

qstring shall be a character string. If the region of the design identified by this cell is an occurrence of a physical primitive from an ASIC library, then *qstring* shall be the name by which the cell is known in the library.

Example (13):

```
(CELLTYPE "DFF")
```

In Example (13), the cell identifies an occurrence of a cell which has the name "DFF" (perhaps a D-type flip-flop).

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

Example (14):

```
(CELLTYPE "TOP")
```

In Example (14), the cell identifies a user-created design block which the user has named "TOP".

If the cell 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* shall be the name by which the modeling primitive is known in that tool.

Example (15):

```
(CELLTYPE "buf")
```

In Example (15), the cell identifies a "buf" modeling primitive in an analysis tool, perhaps a buf "gate" in a Verilog model.

5.3.2 Cell instance

The cell instance shall identify the region or scope of the design for which the cell contains timing data. The name by which this region is known in the design shall be consistent with the **CELLTYPE** for the cell. If the annotator locates the region and finds that its name does not match the **CELLTYPE**, it shall indicate an error. The cell instance has the form given in Syntax 16.

cell_instance ::=
  (INSTANCE [*hierarchical_identifier*] **)**
  | **(INSTANCE *)**

Syntax 16: Syntax for cell instance

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

The hierarchical path shall start at the level in the design at which the annotator shall 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 *hierarchical_identifier*.

Example (16):

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

In Example (16), 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 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 shall apply only to the identified region of the design. If *hierarchical_identifier* is not specified, the default shall be the region (hierarchical level) in the design at which the annotator is instructed to apply the SDF file (see 4.5.2). This can be useful for gathering all interconnect information into a top-level cell.

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

Cells using the wildcard cell instance specification are processed in sequence just like any other cells. No special action shall be taken to consolidate data in this cell with cells with the same cell type earlier or later in the file.

Example (17):

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

In Example (17), every DFF cell instances shall receive the timing data. Note, however, that only cells contained within the region to which the annotator is instructed to apply the SDF file shall be affected.

5.3.3 Timing specifications

Each cell in the SDF file shall include zero or more timing specifications that contain the actual timing data associated with that cell. There are four types of timing specifications that are identified by the **DELAY**, **TIMINGCHECK**, **TIMINGENV**, and **LABEL** keywords. Syntax 17 describes the syntax for timing specification.

```
timing_spec ::=
          del_spec
        | tc_spec
        | lbl_spec
        | te_spec
del_spec ::=
          ( DELAY deltype { deltype } )
tc_spec ::=
          ( TIMINGCHECK tchk_def { tchk_def } )
te_spec ::=
          ( TIMINGENV te_def { te_def } )
lbl_spec ::=
          ( LABEL lbl_type { lbl_type } )
```

Syntax 17: Syntax for timing specification

The **DELAY** keyword shall introduce delays that contain delay data and narrow-pulse propagation data for back-annotation. Delays are described in 5.4.

The **TIMINGCHECK** keyword shall introduce timing checks that contain timing check limit data for back-annotation. Timing checks are described in 5.5.

The **LABEL** keyword shall set the values of timing model variables upon that delays and timing constraint values depend. Labels are described in 5.6.

The **TIMINGENV** keyword shall introduce timing environments that contain timing environment data and constraint data for forward-annotation. Timing environments are described in 5.7.

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

5.4 Delays

Timing specifications that start with the **DELAY** keyword shall 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. The syntax for delay specification is given in Syntax 18.

```
del_spec ::=
          ( DELAY deltype { deltype } )
```

Syntax 18: Syntax for delay specification

One or more *deltype*s can appear in *del_spec*. Each *deltype* shall be **PATHPULSE** or **PATHPULSEPERCENT**, specifying how pulses shall propagate across paths in this cell, or **ABSOLUTE** or **INCREMENT** delay definitions, containing delay values to be applied to the region identified by the cell. Syntax 19 describes the syntax for delay type.

```
deltype ::=
            absolute_deltype
        |   increment_deltype
        |   pathpulse_deltype
        |   pathpulsepercent_deltype
```

Syntax 19: Syntax for delay type

5.4.1 Specifying delay values

In Syntax 25, each construct uses *delval_list* to specify the operating delay values to be applied. A.1.4 provides a formal definition of *delval_list* along with related syntax constructs. Below, *delval_list* is discussed in the context of specifying delay and pulse control data for the various delay constructs in SDF.

The delay data in each delay definition is specified in a list of *delval* tokens. The syntax for delay value list is given in Syntax 20.

```
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 ]
```

Syntax 20: Syntax for delay values list

The number of *delval* tokens in the *delval_list* can be one, two, three, six or twelve. The interpretation of the positional delay values varies with the length of the list. The semantics of *delval_list* for each possible number of *delval* tokens shall be as follows:

— If twelve *delval* tokens are specified in *delval_list*, then each shall correspond, in sequence, to the delay value applicable when the port (output port for **IOPATH** and **INTERCONNECT**) 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* tokens are specified in *delval_list*, then the delays for each transition of the port shall be found from the values given in Table 1. For the column denoting three *delval* tokens in a *delval_list*, the ?→Z transition refers to the third delval token, which represents the delay to Z from both the 0 and 1 states.

Table 1—Deriving 12 delay values

	Number of *delval* tokens in *delval_list*		
Transition	**2**	**3**	**6**
0→1	0→1	0→1	0→1
1→0	1→0	1→0	1→0
0→Z	0→1	?→Z	0→Z
Z→1	0→1	0→1	Z→1
1→Z	1→0	?→Z	1→Z
Z→0	1→0	1→0	Z→0
0→X	0→1	min(0→1, ?→Z)	min(0→1, 0→Z)
X→1	0→1	0→1	max(0→1, Z→1)
1→X	1→0	min(1→0, ?→Z)	min(1→0, 1→Z)
X→0	1→0	1→0	max(1→0, Z→0)
X→Z	max(0→1, 1→0)	?→Z	max(0→Z, 1→Z)
Z→X	min(0→1, 1→0)	min(0→1, 1→0)	min(Z→0, Z→1)

— If only two *delval* tokens are specified, the first (rising) is denoted in Table 1 by 01 and the second (falling) by 10.
— If three *delval* tokens are specified, the first and second are denoted in Table 1 and the third, the Z transition value, by –Z.
— If six *delval* tokens are specified, they are denoted, in sequence, by 0→1, 1→0, 0→Z, Z→1, 1→Z and Z→0 in Table 1.
— If a single *delval* token is specified, it shall apply to all twelve possible transitions. This is not shown in Table 1.

In a *delval_list*, any *delval* tokens can be null, that is, the parentheses enclosing the *signed_real_number* or *rtriple* are empty (see A.1.4). The meaning of this is the same as missing numbers in an *rtriple*: no data is supplied and values shall not be changed by the annotator. Such null *delval* tokens act as placeholders to allow specification of *delval* tokens further down the list.

In a *delval_list* consisting of six or twelve *delval* tokens, the trailing *delval* tokens can be omitted, in which case the omitted values are interpreted as if they were present but null. For example, a list of four *delval* tokens 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* tokens would not be interpreted as a *delval_list* of six *delval* tokens with the three trailing *delval* tokens omitted, since the three remaining *delval* tokens would instead be interpreted as a *delval_list* of three *delval* tokens, as provided in Table 1.

Example (18):

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

In Example (18), 0→1 and 1→0 delay values are not specified and may not even be present in the timing model. A *delval_list* consisting of nothing but null *delval* tokens is permitted by the syntax and shall have no effect.

Each *delval* is either an *rvalue* or a group of two or three *rvalue*s enclosed in parentheses. The syntax for delay value has the form given in Syntax 21.

```
delval ::=
          rvalue
      |   ( rvalue rvalue )
      |   ( rvalue rvalue rvalue )
```

Syntax 21: Syntax for delay value

When a single *rvalue* is used, it specifies the delay value, the pulse rejection limit (r-limit), and the X filter limit (e-limit). When two *rvalue*s in parentheses are used, the first *rvalue* specifies the delay, and the second specifies both the r-limit and the e-limit. When three *rvalues* are used, the first specifies the delay, the second specifies the r-limit, and the third specifies the 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. It provides an advantage over the PATHPULSE and PATHPULSEPERCENT methods of annotating pulse limit value, because both the delays and pulse limits can be specified in a single construct. Pulse limits are described in greater detail in 5.4.2

Each *rvalue* shall be either a single *signed_real_number* or an *rtriple*, containing three *signed_real_number*s separated by colons, in parentheses. The syntax for *rvalue* is shown in Syntax 22.

```
rvalue    ::=
          ( [ signed_real_number ] )
      |   ( [ rtriple ] )
```

Syntax 22: Syntax for rvalue

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

The use of triples in SDF allows 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 two (but not all three) of the numbers in a triple can be omitted if the separating colons are left in place. This shall indicate that no value has been computed for that data, and the annotator shall 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.

NOTE—The amount of data included in a delay definition shall be consistent with the ability of the analysis tool 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, it is inappropriate 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.

5.4.2 Pulse limits

Every delay can have associated with it both a pulse rejection limit (r-limit), and an X filter limit (e-limit). By default, SDF annotation shall set the r-limit and the e-limit equal to the delay. Only an explicit annotation to the contrary can set these values to something other than the delay. Inertial delays are when both limits are equal to the delay, while transport delays are when both limits are equal to zero.

Before a pulse is scheduled to emerge from a port, the pulse is checked to see if it violates either the r-limit or the e-limit. The r-limit is a lower value on the permitted width of the pulse. Any pulse narrower than this limit is rejected, and no pulse will emerge from the port. The e-limit is also a lower value on the permitted width of a pulse, except that instead of being rejected the pulse is filtered to X, unless, of course, the width is also smaller than the r-limit, in which case it is rejected. Figure 4 shows how a pulse scheduled to emerge from a port is checked against both the r-limit and the e-limit.

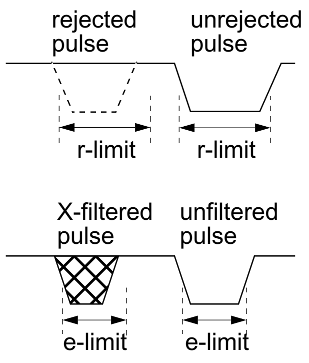


Figure 4—Pulse limits (r-limit and e-limit)

Figure 5 shows how pulses are filtered when the r-limit is 13 and the e-limit is 21. The first pulse presented to the output port, being shorter than 13, shall be rejected. The second pulse, being at least 13, but shorter than 21, shall appear at the output as an X. The third pulse, being at least 21, shall be passed through the output unfiltered.

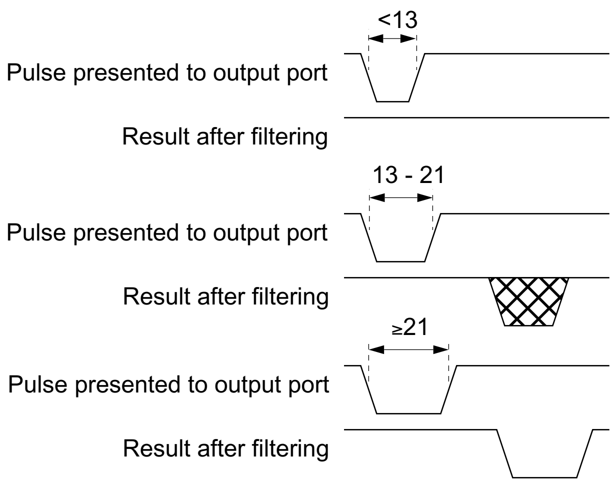
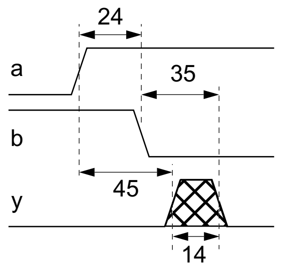

<13
Pulse presented to output port
Result after filtering
13 - 21
Pulse presented to output port
Result after filtering
≥21
Pulse presented to output port
Result after filtering

Figure 5—Pulse filtering

The leading and trailing edges of a pulse at an output port may be due to delays from different input ports, and it is the pulse limits associated with the delay used for the trailing pulse edge that shall be used for determining whether the pulse is passed, filtered to X, or rejected.

Figure 6 shows a 2-input AND gate with rise/fall delays, r-limits, and e-limits. Both inputs are simultaneously high for only a very short period of time. One transitions high just before the other transitions low. Because of differences in the path delays for a rise transition from `a` to `y` and for a fall transition 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 causing the trailing pulse edge, and so its r-limit and e-limit are the ones that apply. The pulse presented to the output is 14 time units, which is larger than the r-limit of 13 for the `b` to `y` path, so the pulse is not rejected. But it is smaller than the e-limit of 21, and so the pulse is filtered to X.



a → y: delay=45/37; r-limit=15; e-limit=24
b → y: delay=43/35; r-limit=13; e-limit=21

Figure 6—2-input AND

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

5.4.3 Absolute delays

The **ABSOLUTE** keyword shall introduce delay data to replace existing delay values in the design during annotation. The syntax for absolute delay is given in Syntax 23.

absolute_deltype ::=
  (ABSOLUTE *del_def* { *del_def* } **)**

Syntax 23: Syntax for absolute delay

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

Example (19):

```
(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 can make use of the negative delays, and for those that cannot the SDF annotator shall set them to zero.

5.4.4 Increment delays

The **INCREMENT** keyword shall introduce delay data that is added to existing delay values in the design during annotation. The syntax for increment delay is described in Syntax 24.

increment_deltype ::=
(INCREMENT *del_def* { *del_def* } **)**

Syntax 24: Syntax for the increment delay

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

Example (20):

```
(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, the value existing in the design shall be reduced. If any negative increment results in negative delays, note that not all analysis tools can make use of negative delays and may set them to zero.

5.4.5 Specifying delays

Both absolute and increment delays shall be described by the same group of delay definition constructs. The syntax for a delay definition is shown in Syntax 25.

```
del_def ::=
        iopath_def
    |   cond_def
    |   condelse_def
    |   port_def
    |   interconnect_def
    |   netdelay_def
    |   device_def
iopath_def ::=
        ( IOPATH port_spec port_instance { retain_def } delval_list )
retain_def ::=
        ( RETAIN retval_list )
cond_def ::=
        ( COND [ qstring ] conditional_port_expr iopath_def )
condelse_def ::=
        ( CONDELSE iopath_def )
port_def ::=
        ( PORT port_instance delval_list )
interconnect_def ::=
        ( INTERCONNECT port_instance port_instance delval_list )
netdelay_def ::=
        ( NETDELAY net_spec delval_list )
device_def ::=
        ( DEVICE [ port_instance ] delval_list )
```

Syntax 25: Syntax for delay definition

5.4.6 Input-output path delays

The input-output path delays shall represent the delays on a legal path from an input/bidirectional port to an output/bidirectional port of a device. Each delay value shall be associated with a unique input port-output port pair. The syntax for input-output path delay from Syntax 25 is given in Syntax 26.

```
iopath_def ::=
        ( IOPATH port_spec port_instance { retain_def } delval_list )
retain_def ::=
        ( RETAIN retval_list )
```

Syntax 26: Syntax for input/output path delay

The fields shall be interpreted as follows:

- The input-output path delay shall be specified using the keyword **IOPATH**.
- *port_spec* shall be an input or a bidirectional port and can have an edge identifier.
- *port_instance* shall be an output or a bidirectional port. It cannot have an edge identifier. Delay data for the different transitions at the path output port shall be supplied using an ordered list of delay values as described in Table 1 (see 5.4.1).
- *retain_def* is discussed in 5.4.9.
- *delval_list* shall specify the delay data from *port_spec* to *port_instance*.

This form of **IOPATH** has no conditions (state dependency) associated with it, and so it annotates independent of the conditions defined within timing models. If the timing model includes conditions for the path delay between the two specified ports, the specified *delval* shall still applied. If the model includes more than one delay path, each distinguished by its condition, then the data shall apply to all of them. This shall have 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*.

The only exception is when *port_spec* includes an edge identifier, in which case **IOPATH** will only annotate when the timing model includes an equivalent edge identifier. See 5.5.2 for similar information regarding timing checks.

Example (21):

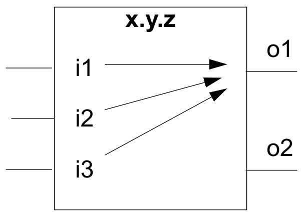


Figure 7—Input/output path delay

```
    (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))
       )
    )
```

The **IOPATH** construct can also be used to specify the r-limit and e-limit for each transition delay.

Example (22):

```
    (INSTANCE x.y.z)
    (DELAY
       (ABSOLUTE
          (IOPATH (posedge i1) o1 ((12:25:37) (5:12:17)) )
          (IOPATH i2 o1 ((4:6:8) (2:3:4) (4:5:6)) ((5:7:9) (3:4:5)
(5:6:7)) )
          (IOPATH i3 o1 () () (2:4:5) (4:5:6) (5:6:7) (7:8:9))
       )
    )
```

In Example (22), the first **IOPATH** specifies a min/typ/max limit that sets both the r-limit and the e-limit for all transition delays. All delays are set using the 12:25:37 triple, while all r-limits and e-limits are set using the 5:12:17 limit.

The second **IOPATH** specifies min/typ/max r-limits and e-limits for the rise and fall transition delays. The rise delay is set using the 4:6:8 triple, the rise r-limit is set using the 2:3:4 triple, and the rise e-limit is set using the 4:5:6 triple. The fall delay is set using the 5:7:9 triple, the fall r-limit is set using the 3:4:5 triple, and the fall e-limit is set using the 5:6:7 triple.

The third **IOPATH** specifies no pulse limits at all, and so the r-limits and e-limits for each transition delay default to the delay values. The rise delay and its r-limit and e-limit are unchanged. The same is true for the fall delay. The 0→Z delay and its r-limit and e-limit are set using the 2:4:5 triple. The Z→1 delay and its r-limit and e-limit are set using the 4:5:6 triple. The 1→Z delay and its r-limit and e-limit are set using the 5:6:7 triple. And the Z→0 delay and it's r-limit and e-limit are set using the 7:8:9 triple.

5.4.7 Conditional path delays

The conditional path delay shall specify conditional (state-dependent) input-to-output path delays. The syntax for conditional path delay from Syntax 25 is given in Syntax 27.

cond_def ::=
(COND [*qstring*] *conditional_port_expr iopath_def* **)**

Syntax 27: Syntax for conditional path delay

The fields shall be interpreted as follows:

— The conditional path delay shall be specified using the keyword **COND**.
— *qstring* shall be an optional symbolic name as a placeholder for annotators that operate by matching named placeholders in the model to SDF constructs. See 5.4.8 for a full explanation.
— *conditional_port_expr* shall be the description of the state dependency of the path delay. The syntax of *conditional_port_expr* is shown in A.1.5. The expression shall evaluate to a logical signal, rather than a boolean. The analysis tool shall treat a logical zero as FALSE and any other logical value (1, X, or Z) as TRUE. A particular conditional path delay in the timing model shall be used only if the condition is TRUE.
— *iopath_def* shall have the same meaning as in **IOPATH** as described in 5.4.6, except that the annotator shall 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 shall not receive the delay data.
— Annotators can 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, the annotator shall require an exact character-for-character match in the string representations of the conditions.

The annotator must locate in the timing model a path delay with conditions matching those specified in the SDF file. Other path delays between the same ports but with different conditions shall not receive the data. SDF **IOPATH** constructs with no conditions match any path delay in the model that is between the ports specified in the SDF file.

Example (23):

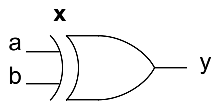


Figure 8—Conditional path delay

```
(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 shall specify default delays for conditional paths. The default delay is the delay that shall be used if none of the conditions specified for the path in the model are TRUE but a signal must still be propagated over the path. The syntax for default delay for conditional path from Syntax 25 is given in Syntax 28.

condelse_def ::=
    (CONDELSE *iopath_def* **)**

Syntax 28: Syntax for default delay for conditional path

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

5.4.8 Condition labels

Some annotators, particularly those annotating to VITAL models in VHDL simulators, automatically transform the names used in SDF constructs into symbolic names using a set of simple rules, locate those symbolic names within the analysis tool models, and apply values from the SDF file to the variables associated with those symbolic names. The automatic transformation rules cannot guarantee that unique names will always result. This problem is most commonly encountered in situations involving conditional expressions, and for this reason conditional expressions can be optionally preceded by a *qstring*.

A tool can use the *qstring* symbolic name directly, or it can modify it in a clearly documented way. For example, if the *qstring* was `tdtoq` and appeared with an **IOPATH** construct, a tool might use the *qstring* by itself as the symbolic name and look up `tdtoq`. Alternatively, the tool might precede it with the name of the construct and instead look up `IOPATH_tdtoq`. The mapping algorithm of the tool's annotator must be clearly documented and available to users, and must include the way in which the *qstring* is used in constructing the name. The description should also explain what happens if the *qstring* is absent in a conditional construct and what happens in certain timing checks where a match with two or more *qstring*s is possible.

Associating a *qstring* with conditional expressions is only a partial solution, since even symbolic names associated with unconditional **IOPATH** delays and timing checks can be non-unique, and so a more general approach using the **LABEL** construct, described in 5.6, is recommended.

5.4.9 Output retain delays

In an **IOPATH**, the keyword **RETAIN** shall specify the time for which an output/bidirectional port shall retain its previous logic value after a change at a related input/bidirectional port. **RETAIN** is commonly used on paths from the address or select inputs to the data outputs of memory and register file circuits. The syntax for output retain delay is described in Syntax 26.

The keyword **RETAIN** shall specify the delay data used for retaining the value of the output or bidirectional port. Delay data for different transitions at the path output port are supplied using the *retval_list* ordered list of values, a truncated form of *delval_list* (described in 5.4.1) that can have up to three delays. The retain delay shall be specified only as part of input-output path delay, conditional path delay or default delay for a conditional path.

This construct shall be used only where the cell timing model being annotated is capable of providing retain delays. The annotator shall match this SDF construct to such a mechanism in the model.

Example (24):

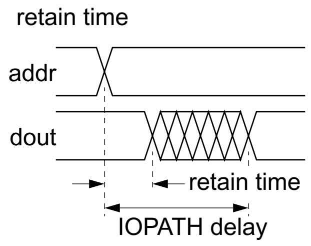


Figure 9—Output retain delay

```
    (IOPATH addr[ 13:0]  dout[ 7:0]
       (RETAIN (4:5:7) (5:6:9))    // RETAIN delays
       (15:20:25) (18:22:27))      // IOPATH delays
)
```

Example (24) includes the retain time of bus `dout[ 7:0]` with respect to changes on the bus `addr[ 13:0]` . It is assumed that the timing model for this cell contains path delays from `addr` to `dout` and is capable of handling retain delays. In response to a transition on bus `addr`, output `dout` will transition to the X state. The first **RETAIN** *delval*, `(4:5:7)`, is the rise time and shall be used for `dout` going from low to X. The second **RETAIN** *delval*, `(5:6:9)`, is the fall time and shall be used for `dout` going from high to X. Output `dout` will next transition from X to it's final state. The first **IOPATH** *delval*, `(15:20:25)`, shall be used when `dout` transitions from X to low. The second **IOPATH** *delval*, `(18:22:27)`, shall be used when `dout` transitions from X to high.

5.4.10 Port delays

The port delays shall specify 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. The syntax for a port delay from Syntax 25 is given again in Syntax 29.

port_def ::=
(**PORT** *port_instance delval_list* **)**

Syntax 29: Syntax for port delay

The fields shall be interpreted as follows:

— The port delay shall be specified using the keyword **PORT**.
— *port_instance* shall be an input or bidirectional port.
— *delval_list* shall be the port delay of the *port_instance*.

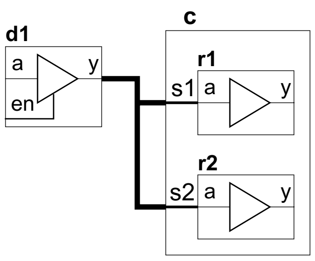


Figure 10—Port delay

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

5.4.11 Interconnect delays

The interconnect delay shall specify the propagation delay across a net connecting a driving module port (the source) to a driven module port (the load). Either or both ports can be bidirectional. Both source and load ports for the delay path shall be specified. The syntax for interconnect delay from Syntax 25 is given again in Syntax 30.

interconnect_def ::=
(**INTERCONNECT** *port_instance port_instance delval_list* **)**

Syntax 30: Syntax for interconnect delay

The fields shall be interpreted as follows:

— The interconnect delay shall be specified using the keyword **INTERCONNECT**.
— The first *port_instance* shall be an output or bidirectional port (driver port).
— The second *port_instance* is an input or bidirectional port (driven port).
— *delval_list* shall be the interconnect delay between the output and input ports.

Example (25):

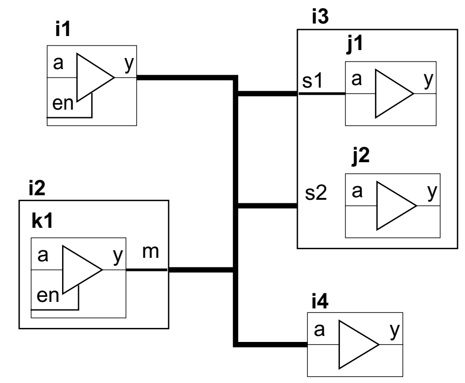


Figure 11—Interconnect delay

```
(INSTANCE top)
(DELAY
   (ABSOLUTE
      (INTERCONNECT  i1.y     i3.j1.a  (0.01:0.02:0.03))
      (INTERCONNECT  i1.y     i3.j2.a  (0.03:0.04:0.05))
      (INTERCONNECT  i1.y     i4.a     (0.05:0.06:0.07))
      (INTERCONNECT  i2.k1.y  i3.j1.a  (0.04:0.05:0.06))
      (INTERCONNECT  i2.k1.y  i3.j2.a  (0.02:0.03:0.04))
      (INTERCONNECT  i2.k1.y  i4.a     (0.02:0.03:0.04))
   )
)
```

Although **INTERCONNECT** constructs 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 shall map interconnect delays into equivalent input port delays, 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 shall bear this in mind when choosing whether to use **PORT** or **INTERCONNECT** constructs or a combination of both to model interconnect delay.

5.4.12 Net delays

The net delays shall specify the propagation delays from all sources to all loads of a net. Neither start nor end points for the delay path are specified, and the delays from all the source ports to all the destination ports will have the same value. Using a **NETDELAY** entry would be the same as using **INTERCONNECT** delay entries for each source/load pair on the net and giving them all the same value. The syntax for a port delay from Syntax 25 is given again in Syntax 31.

netdelay_def ::=
        (NETDELAY *net_spec delval_list* **)**

Syntax 31: Syntax for net delay

Example (26):

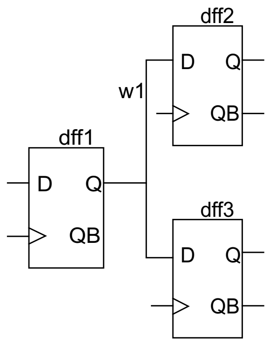


Figure 12—Net delay

```
(INSTANCE x)
(DELAY
   (ABSOLUTE
      (NETDELAY w1 (2.5:3:3.5) (2.9:4:5))
   )
)
```

In Example (26), the net has been identified by name. It could also have been identified by any of the ports attached to it.

5.4.13 Device delays

The device delay shall represent the delay of all paths through a cell to the specified output port. This construct shall be used primarily 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 shall be same as applying the delay data to all input-to-output paths across the cell that terminate at the specified port. If there are no path delays to the port, then the delay is annotated to all gate primitives driving the port, regardless of their actual hierarchical level within the model. The syntax for a device delay from Syntax 25 is shown in Syntax 32.

device_def ::=
    (DEVICE [*port_instance*] *delval_list* **)**

Syntax 32: Syntax for device delay

The fields shall be interpreted as follows:

— The device delay shall be specified using the keyword **DEVICE**.
— *port_instance* shall be optional. If present, it shall specify the output port to which the delay data shall be applied. If a cell has more than one output, several device delays in a single cell can be specified, each indicating the desired output port using *port_instance*, and with a different delay data to each output. If *port_instance* is omitted, all paths to all output ports of the region identified in the cell shall receive the same delay data.
— *delval_list* shall be the delay data. The number of *triple*s in *delval_list* shall correspond to the capabilities of the modeling primitives of the target analysis tool. For example, gate level primitives in Verilog HDL can accept one, two, or in some cases, three delay values, but never six or twelve.

Example (27):

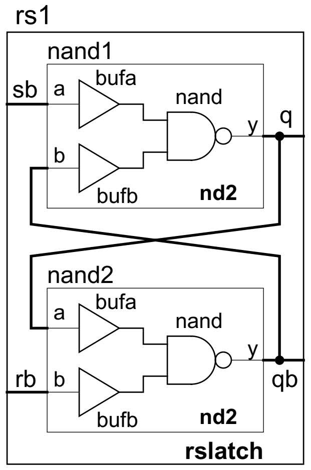


Figure 13—RS latch in distributed delay style

```
(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 Example (27), an RS latch is implemented from two `nd2` macrocells. An `nd2` is a 2-input NAND macrocell constructed in a distributed delay style from two BUF primitives, `bufa` and `bufb`, and a NAND primitive, `nand`. Two `nd2` macrocells are cross-coupled to create the RS latch, which is given the name `rslatch`. This is instantiated at a higher level of the design as `rs1`. The SDF file demonstrates the annotation of delays to BUF primitives `rs1.nand1.bufa` and `rs1.nand1.bufb`, in effect defining unique delays for the `a-to-y` and `b-to-y` paths. The first annotation has the effect of defining the `sb`-to-`qb` input-to-output path delay 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. This example also makes clear the difficulty of accurately annotating delay values that yield the correct pin-to-pin delay. As presented here, the `sb`-to-`qb` delay is 0. If a delay is annotated to `sb`-to-`qb` to yield the correct `sb`-to-`qb` delay, it will also affect the `sb`-to-`q` delay. Determining the correct delay values to annotate to each gate primitive can be a difficult task, and in many cases no non-imaginary set of delay values satisfy the delay specification of the device.

Example (28):

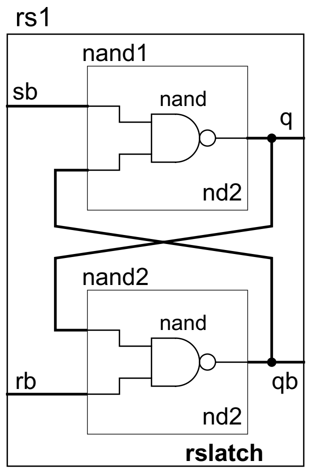


Figure 14—RS latch in a pin-to-pin modeling style

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

Example (28) assumes a path delay modeling style. The same RS latch is described using `nd2` macrocells that contain slightly less detail, as they are no longer implemented using BUF primitives. With no BUF primitives to which to annotate delays, there can be no unique pin-to-pin delays using a distributed delay style. Typically the path delay modeling style defines unique delays for the `sb-to-q`, `sb-to-qb`, `rb-to-q` and `rb-to-qb` paths. This SDF file does not take full advantage of this, however. One delay is annotated to both the `sb-to-q` and `rb-to-q` paths, the other to both the `sb-to-qb` and `rb-to-qb` paths. It shall 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))
      )
   )
)
```

5.4.14 Pathpulse

PATHPULSE specifies pulse propagation limits (the r-limit and the e-limit) associated with a legal path between an input port and an output port of a device. These limits shall determine whether a pulse presented at an output port will emerge unfiltered, be filtered to X, or be rejected.

```
pathpulse_deltype ::=
        ( PATHPULSE [ input_output_path ] value [ value ] )
input_output_path ::=
        port_instance port_instance
```

Syntax 33: Syntax for pathpulse

- The first *port_instance* of *input_output_path* shall be an input or a bidirectional port. The second *port_instance* of *input_ouput_path* shall be an output or a bidirectional port.
- If *input_output_path* is omitted, then the data supplied shall refer to all input-to-output paths in the region identified by the cell. The annotator shall 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, shall be the pulse rejection limit, also known as the r-limit. This limit shall define the narrowest pulse that can appear at the output port of the specified path. Any pulse narrower than the specified value shall not appear at the output, but shall be rejected.
- The second *value*, in time units, shall be the X or error limit, also known as the e-limit. This limit shall define the minimum pulse width necessary to drive the output of the specified path to a known state; a narrower pulse shall cause the output to enter the unknown (X) state or shall be rejected (if smaller than the r-limit). Note that the e-limit shall be greater than the r-limit to carry any significance.

If only one *value* is specified, both limits shall be set to that value. In all cases *value* can be either a single number or a *triple*, but shall not be negative.

Example (29):

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

Using the **PATHPULSE** construct is more verbose and takes more time to annotate than expressing the pulse limits more compactly using the **IOPATH** construct.

Example (30):

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

An additional advantage of the **IOPATH** construct is that it can specify unique pulse limits for each transition delay. **PATHPULSE** can only annotate a single r-limit/e-limit value pair across all transition delays.

Example (31):

```
(INSTANCE x)
(DELAY
   (ABSOLUTE
      (IOPATH a y ((45) (13) (24)) ((37) (11) (19)) )
      (IOPATH b y ((43) (14) (20)) ((35) (10) (17)) )
   )
)
```

5.4.15 Pathpulsepercent

PATHPULSEPERCENT shall have the same interpretation as **PATHPULSE** except that the values are expressed as a percentage of the cell path delay from the input to the output.

pathpulsepercent_deltype ::=
(PATHPULSEPERCENT [*input_output_path*] *value* [*value*] **)**

Syntax 34: Syntax for pathpulsepercent

Neither *value* shall be greater than 100.0. As discussed in 5.4.14, the second *value* (e-limit) shall be greater than the first *value* (r-limit) for it to be meaningful.

Example (32):

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

In Example (32), the r-limit is specified as 25% of the delay time from a to y and the e-limit is specified as 35% of this delay. If more than one *delval* is specified in the *delval_list* of **IOPATH**, the analysis tool shall select that corresponding del_val to the transition that ended the pulse. So, for a low-to-high transition 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 previous example, where the high-to-low delay is 37, the r-limit is 25% of 37 and the e-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).

5.5 Timing checks

Timing specifications that start with the **TIMINGCHECK** keyword shall associate timing check limit values with specific cell instances. The syntax for timing specification is described in Syntax 35.

```
tc_spec ::=
        ( TIMINGCHECK tchk_def { tchk_def } )
```

Syntax 35: Syntax for timing specification

Any number of *tchk_def* constructs can appear in a *tc_spec*. Each *tchk_def* shall be one of the following timing checks, containing timing check limit values as defined in Syntax 36:

Timing checks 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 shall determine the constraints for those paths.
— Synthesis tools use timing check values to determine if their results meet timing requirements.

The syntax for timing checks is given in Syntax 36.

tchk_def ::=
    setup_timing_check
  | *hold_timing_check*
  | *setuphold_timing_check*
  | *recovery_timing_check*
  | *removal_timing_check*
  | *recrem_timing_check*
  | *skew_timing_check*
  | *bidirectskew_timing_check*
  | *width_timing_check*
  | *period_timing_check*
  | *nochange_timing_check*
setup_timing_check ::=
    (SETUP *port_tchk port_tchk value* **)**
hold_timing_check ::=
    (HOLD *port_tchk port_tchk value* **)**
setuphold_timing_check ::=
    (SETUPHOLD *port_tchk port_tchk rvalue rvalue* **)**
  | **(SETUPHOLD** *port_spec port_spec rvalue rvalue* [*scond*] [*ccond*] **)**
recovery_timing_check ::=
    (RECOVERY *port_tchk port_tchk value* **)**
removal_timing_check ::=
    (REMOVAL *port_tchk port_tchk value* **)**
recrem_timing_check ::=
    (RECREM *port_tchk port_tchk rvalue rvalue* **)**
  | **(RECREM** *port_spec port_spec rvalue rvalue* [*scond*] [*ccond*] **)**
skew_timing_check ::=
    (SKEW *port_tchk port_tchk rvalue* **)**
bidirectskew_timing_check ::=
    (BIDIRECTSKEW *port_tchk port_tchk value value* **)**
width_timing_check ::=
    (WIDTH *port_tchk value* **)**
period_timing_check ::=
    (PERIOD *port_tchk value* **)**
nochange_timing_check ::=
    (NOCHANGE *port_tchk port_tchk rvalue rvalue* **)**

Syntax 36: Syntax for timing checks

5.5.1 Conditional timing checks

The **COND** keyword shall allow the specification of conditional timing checks. Its use is different from the specification of conditional input-output path delays described in 5.4.7 in that the condition is associated with the specification of a port rather than the entry as a whole. The syntax for conditional timing check is given in Syntax 37.

port_tchk ::=
    port_spec
  | **(COND** [*qstring*] *timing_check_condition port_spec* **)**

Syntax 37: Syntax for conditional timing check

qstring shall be an optional symbolic name placeholder for annotators that operate by matching named placeholders in the model to SDF constructs. See 5.4.8 for detailed explanation.

timing_check_condition is the description of the state dependency of the timing check. The syntax of *timing_check_condition* is shown in A.1.6. This expression shall evaluate to a logical signal, rather than a boolean. The analysis tool shall treat a logical zero as FALSE and any other logical value (1, X, or Z) as TRUE. A particular conditional timing check in the timing model shall be used only if the condition is TRUE.

The annotator must locate in the timing model a timing check with conditions matching those specified in SDF file. Other timing checks of the same kind but with different conditions shall not receive the data. SDF timing checks 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 can be used 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*. A stamp event defines the beginning of a measured interval, and a check defines the end of a measured interval. Separate conditions can be supplied for the stamp and check events using the **SCOND** and **CCOND** keywords. **SCOND** or **CCOND** or both **SCOND** and **CCOND** shall take precedence over **COND.**

The syntax for stamp condition and check condition is given in Syntax 38.

scond ::=
    (SCOND [*qstring*] *timing_check_condition* **)**
ccond ::=
    (CCOND [*qstring*] *timing_check_condition* **)**

Syntax 38: Syntax for stamp and check conditions

For the setup phase of a setuphold timing check, the stamp condition shall apply to the data port and the check condition to the clock or gate port. For the hold phase, the stamp condition shall apply 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 shall 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 second event):

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

5.5.2 Edge Specifications

Any *port_spec* can be qualified with an edge identifier as it is given in Syntax 39.

```
port_spec ::=
        port_instance
    |   port_edge
port_edge ::=
        ( edge_identifier port_instance )
```

Syntax 39: Syntax for port specification

A port specification with an edge identifier is called an *edge specification*. When the annotator is locating a timing check at specified ports in the timing model, it shall match the edge specification as well as the port names. A port without an edge specification in SDF shall match all edge specifications in the model.

Example (33):

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

Example (33) shows a cell with setup and hold timing checks specified on data port `din` with respect to the rising edge of the clock signal.

5.5.3 Specifying timing check limit values

In the syntax descriptions of the timing check constructs, either *rvalue* or *value* is used to specify the timing check limit. The *rvalue* can be negative, zero, or positive, and negative values are only legal in **SETUPHOLD**, **RECREM**, and **NOCHANGE** constructs. The *value* shall be zero or positive.

Each *rvalue* or *value* shall be a single value (*signed_real_number* or *real_number*, respectively) or three values separated by colons (an *rtriple* or *triple,* respectively), representing three sets of data for minimum, typical, and maximum delay conditions.

The use of triples in SDF allows 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 shall be omitted if the separating colons are left in place. This shall indicate that no value has been computed for that data, and the annotator shall not make any changes if that number is selected from the triple.

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

5.5.4 Setup timing checks

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 shall define the part of the interval before the clock transition; the hold time limit shall define the part of the interval after the clock transition. Any change to the data signal within this interval shall result 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 shall always remain greater than zero. Syntax 40 gives the syntax for a setup timing check.

setup_timing_check ::=
  (SETUP *port_tchk port_tchk value* **)**

Syntax 40: Syntax for setup timing check

The fields shall be interpreted as follows:

— The timing check beginning with keyword **SETUP** shall specify limit values for a setup timing check.
— The first *port_tchk* shall identify the data port. If it includes an edge specification, then the *value* shall be used for a setup timing check with respect to only the specified transition at the data port.
— The second *port_tchk* shall identify the clock or gate port and shall normally include an edge specification to identify the active edge of the clock or the active-to-inactive transition of the gate.
— *value* shall be the setup time limit between the data and clock ports and shall not be negative.

Example (34):

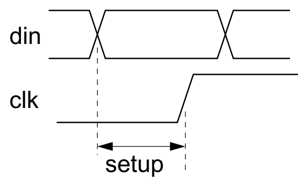


Figure 15—Setup time

```
(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.

5.5.5 Hold timing checks

The syntax for hold timing check is given in Syntax 41.

hold_timing_check ::=
(HOLD *port_tchk port_tchk value* **)**

Syntax 41: Syntax for hold timing check

The fields shall be interpreted as follows:

— The timing check beginning with keyword **HOLD** shall specify limit values for a hold timing check.
— The first *port_tchk* shall identify the data port.
— The second *port_tchk* shall identify the clock port.
— *value* shall be the **HOLD** time between the data and clock events and shall not be negative.

See 5.5.4 for a description of the use of hold timing checks and more information about the use of edge specifications in this context.

Example (35):

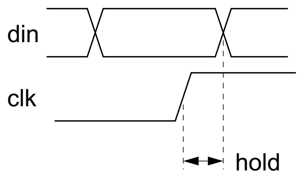


Figure 16—Hold time

```
(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.

5.5.6 SetupHold timing checks

Syntax 42 describes the syntax for setuphold timing check.

setuphold_timing_check ::=
(SETUPHOLD *port_tchk port_tchk rvalue rvalue* **)**
| **(SETUPHOLD** *port_spec port_spec rvalue rvalue* [*scond*] [*ccond*] **)**

Syntax 42: Syntax for setuphold timing check

The fields shall be interpreted as follows:

— The timing check beginning with the keyword **SETUPHOLD** shall specify setup and hold limits in a single timing check.
— The first *port_tchk* or *port_spec* shall identify the data port.
— The second *port_tchk* or *port_spec* shall identify the clock port.

— The first *rvalue* shall be the setup time and the second *rvalue* shall be the hold time. Either can be negative, but the sum of the two *rvalues* shall be greater than zero.
— The optional *scond* and *ccond* are the stamp and check conditions as described in 5.5.1.

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.

See 5.5.4 for the use of setup and hold timing checks and edge specifications in this context.

Example (36):

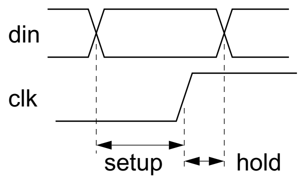


Figure 17—Setup and hold time

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

This SDF entry shall 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 (37):

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

This SDF entry, using the second syntax form, shall 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 shall be when the `clk` port undergoes a **posedge** transition. For the hold phase of the check, this shall be when the `din` port undergoes any transition.

5.5.7 Recovery timing checks

The **RECOVERY** shall specify 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 shall 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. The syntax for recovery timing check is given in Syntax 43.

recovery_timing_check ::=
  (RECOVERY *port_tchk port_tchk value* **)**

Syntax 43: Syntax for recovery timing check

— The first *port_tchk* refers to the asynchronous control signal and shall 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 shall 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.

Example (38):

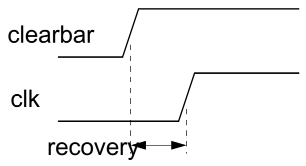


Figure 18—Recovery time

```
(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.

5.5.8 Removal timing checks

The **REMOVAL** shall specify 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 shall 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. The syntax for removal timing check is described in Syntax 44.

removal_timing_check ::=
  (REMOVAL *port_tchk port_tchk value* **)**

Syntax 44: Syntax for removal timing check

— The first *port_tchk* refers to the asynchronous control signal and shall 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 shall 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 shall ignore any normal operation, such as clocking in of new data.

Example (39):

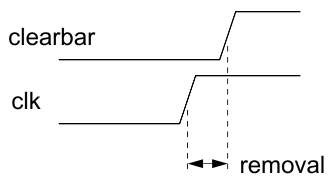


Figure 19—Removal time

```
(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.

5.5.9 Recovery/removal timing checks

The **RECREM** construct shall specify both recovery and removal limits in a single entry. The syntax for recovery/removal timing check is given in Syntax 45.

> *recrem_timing_check* ::=
> **(RECREM** *port_tchk port_tchk rvalue rvalue* **)**
> | **(RECREM** *port_spec port_spec rvalue rvalue* [*scond*] [*ccond*] **)**

Syntax 45: Syntax for recovery/removal timing check

— 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.
— The optional *scond* and *ccond* are the “stamp” and “check” conditions as described in 5.5.1.

Example (40):

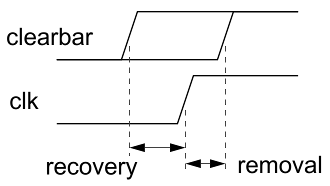


Figure 20—Recovery and removal time

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

Example (40) 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.

5.5.10 Skew timing checks

The **SKEW** construct shall specify limit values for unidirectional 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 46 shows the formal syntax for unidirectional skew timing check.

skew_timing_check ::=
    (SKEW *port_tchk port_tchk rvalue* **)**

Syntax 46: Syntax for skew timing check

— The first *port_tchk* shall be either the stamp or check event, depending upon whether *rvalue* is positive or negative.
— The second *port_tchk* shall be either the stamp or check event, depending upon whether *rvalue* is positive or negative.
— *rvalue* is the maximum skew limit.

The unidirectional **SKEW** construct shall annotate values for skew checks where the skew can only be measured in one direction. If *rvalue* is positive, then skew is measured from the first *port_tchk* to the second. A negative *rvalue* reverses the sense of the check so that the skew is measured from the second *port_tchk* to the first.

Example (41):

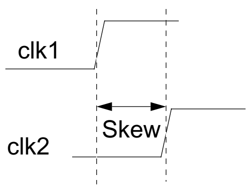


Figure 21—Skew timing check

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

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

5.5.11 Bidirectional skew timing checks

The **BIDIRECTSKEW** construct shall specify limit values for bidirectional 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 47 shows the formal syntax for the bidirectional skew timing check.

bidirectskew_timing_check ::=
  (BIDIRECTSKEW *port_tchk port_tchk value value* **)**

Syntax 47: Syntax for bidirectional skew timing check

The first *port_tchk* shall be either the stamp or check event, depending upon whether it transitions first of second.
The second *port_tchk* shall be either the stamp or check event, depending upon whether it transitions first or second.
value is the maximum skew limit when the first *port_tchk* transitions first.
value is the maximum skew limit when the second *port_tchk* transitions first.

The bidirectional **BIDIRECTSKEW** construct shall annotate values for skew checks where the skew can be measured in either direction. Either port_tchk can transition first. Both limits must be positive.

Example (42):

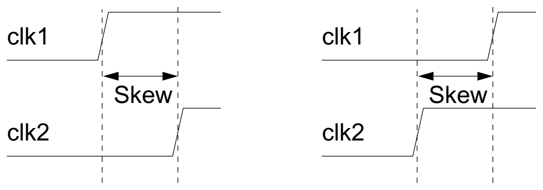


Figure 22—Skew timing check

```
(INSTANCE x)
(TIMINGCHECK
   (BIDIRECTSKEW (posedge clk1) (posedge clk2) (6) (7))
)
```

When `clk1` transitions first, the delay before the `clk2` transition is measured against the first *value*, 6. When `clk2` transitions first, the delay before the `clk1` transition is measured against the second *value*, 7.

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

5.5.12 Width timing checks

The **WIDTH** shall specify 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. The syntax for width timing check is given in Syntax 48.

width_timing_check ::=
  (WIDTH *port_tchk value* **)**

Syntax 48: Syntax for width timing check

— *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 shall 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 (43):

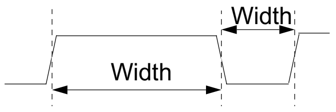


Figure 23—Width timing check

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

In Example (43), 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.

5.5.13 Period timing checks

The **PERIOD** shall specify 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. The syntax for period timing check is given in Syntax 49.

period_timing_check ::=
(PERIOD *port_tchk value* **)**

Syntax 49: Syntax for period timing check

— *port_tchk* refers to the port at which the minimum period timing check is applied. If it includes an edge specification, then the data shall 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 (44):

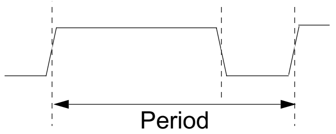


Figure 24—Period timing check

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

In Example (44), 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.

5.5.14 No change timing checks

The **NOCHANGE** shall specify 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. The syntax for any nochange timing check is given in Syntax 50.

nochange_timing_check ::=
(NOCHANGE *port_tchk port_tchk rvalue rvalue* **)**

Syntax 50: Syntax for nochange timing check

— 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 (45):

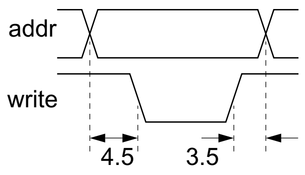


Figure 25—Nochange timing check

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

Example (45) 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.

5.6 Labels

Labels enable the direct annotation of variables such as Verilog specparams and VHDL generics. The use of labels has the potential to improve four aspects of SDF annotation. Labels can be looked up faster than port names and conditions, improving SDF annotation performance. For Verilog, it is the only way to annotate behavioral delays. For VHDL, it provides unambiguous annotation to generics. And new SDF constructs will not be needed when new capabilities are added to Verilog or VHDL or other HDLs, because labels can be used to annotate timing values to the new features.

lbl_spec ::=
  (LABEL *lbl_type* { *lbl_type* } **)**
lbl_type ::=
  (INCREMENT *lbl_def* { *lbl_def* } **)**
  | **(ABSOLUTE** *lbl_def* { *lbl_def* } **)**
lbl_def ::=
  (*identifier delval_list* **)**

Syntax 51: Syntax for label specification

Labels are supported via the **LABEL** construct, which can appear inside a **CELL** construct at the same level as the **DELAY**, **TIMINGCHECK**, and **TIMINGENV** constructs. Label values are expressed as a *delval_list*, and can be either incremental or absolute.

Values in a *delval_list* in excess of what is required by that being annotated are ignored. For example, if there are six values in a *delval_list* that is being annotated to a Verilog gate primitive, then the first three values in the *delval_list* are used and the last three values in the *delval_list* are ignored. SDF annotators can issue a warning when this circumstance occurs.

As already mentioned previously, an SDF file shall contain any number of cells (other than zero). The order of the cells shall be significant only if they have overlapping effect, in other words, if data from two different cells applies to the same timing property in the design. In this situation, the cells shall be processed strictly from the beginning of the file towards the end and the data they contain shall be applied in sequence to whatever region is appropriate to that cell. If data is applied to a timing property previously referenced by an SDF file, the new data shall be applied over the old and the final value shall be the cumulative effect according to whether the data is applied as a replacement for the old value (**ABSOLUTE** section) or is added to it (**INCREMENTAL** section).

Example (46):

SDF file:

```
(CELL (CELLTYPE "DFF")
   (INSTANCE cache.mem3.bndrx.I103)
   (LABEL
      (ABSOLUTE
         (TCLK_Q (7.54:12.14:19.78) (6.97:13.66:18.47))
         (TCLK_QB (8.16:13.74:20.25) (7.63:14.83:21.42))
         (TSETUP_D_CLK (3:4:5.6))
         (THOLD_D_CLK (4:5.6:7))
)  )  )
```

Verilog specify block annotated to by the above SDF:

```
specify
   specparam
      TCLK_Q = 0,
      TCLK_QB = 0,
      TSETUP_D_CLK = 0,
      THOLD_D_CLK = 0;
   (CLK => Q) = TCLK_Q;
   (CLK => QB = TCLK_QB;
   $setuphold (CLK, D, TSETUP_D_CLK, THOLD_D_CLK);
endspecify
```

VITAL model annotated to by the above SDF (only the most relevant portions are shown, and the actual declaration of the generics is not included):

```
VitalPathDelay (Q, "Q", Q_int,
   Paths => (0 => (CLK_ipd’LAST_EVENT, TCLK_Q, TRUE)));
VitalPathDelay (QB, "QB", QB_int,
   Paths => (0 => (CLK_ipd’LAST_EVENT, TCLK_QB, TRUE)));
VitalSetupHoldCheck (D, "D”, CLK, "CLK",
   SetupHigh => TSETUP_D_CLK,
   SetupLow => TSETUP_D_CLK,
   HoldHigh => THOLD_D_CLK,
   HoldLow => THOLD_D_CLK);
```

For **LABEL** constructs, SDF annotators shall enforce all rules concerning negative values. SDF annotators must check that negative **LABEL** values result in legal timing values for all contexts where the label appears in the timing model. Specifically, the SDF annotator would have to detect and issue a warning whenever the label is used in a timing model in such a way so as to yield a negative value where negative values are not legal, such as delays and some timing checks.

5.7 Timing environment

Timing specifications that start with the **TIMINGENV** keyword shall associate constraint values with critical paths in the design and provide information about the timing environment in which the circuit shall operate. Constructs in this subclause are used in forward-annotation and not back-annotation. The syntax for timing environment is shown in Syntax 52.

```
te_spec ::=
        ( TIMINGENV te_def { te_def } )
```

Syntax 52: Syntax for timing environment

Any number of *te_def*s shall appear in a *te_spec*. Each *te_def* shall be a **PATHCONSTRAINT**, **PERIOD-CONSTRAINT**, **SUM**, **DIFF**, or **SKEWCONSTRAINT** constraint, containing constraint values for the design or an **ARRIVAL**, **DEPARTURE**, **SLACK**, or **WAVEFORM** timing environment, containing information about the timing environment in which the circuit shall operate. Syntax 53 describes the syntax for timing definition.

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

Syntax 53: Syntax for timing definition

5.7.1 Constraint constructs

Constraint constructs shall 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.) shall adapt its strategy to try to ensure that the constraints are met and issue warning messages in the event that they cannot be met. The syntax for constraint definition is given in Syntax 54.

```
cns_def ::=
        path_constraint
      | period_constraint
      | sum_constraint
      | diff_constraint
      | skew_constraint
path_constraint ::=
        ( PATHCONSTRAINT [ name ] port_instance port_instance { port_instance }
                rvalue rvalue )
period_constraint ::=
        ( PERIODCONSTRAINT port_instance value [ exception ] )
sum_constraint ::=
        ( SUM constraint_path constraint_path { constraint_path } rvalue [ rvalue ] )
diff_constraint ::=
        ( DIFF constraint_path constraint_path value [ value ] )
skew_constraint ::=
        ( SKEWCONSTRAINT port_spec value )
```

Syntax 54: Syntax for constraint definition

Subclauses 5.7.1.1 through 5.7.1.5 describes the SDF constraint constructs.

5.7.1.1 Path constraints

The **PATHCONSTRAINT** shall represent 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. The syntax for path constraint is shown in Syntax 55.

```
path_constraint ::=
    ( PATHCONSTRAINT [ name ] port_instance port_instance { port_instance }
    rvalue rvalue )
name ::=
        ( NAME qstring )
```

Syntax 55: Syntax for path constraint

name is optional and allows a symbolic name to be associated with the path. This name shall 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 (47):

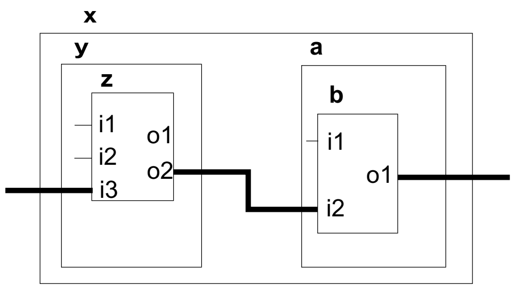


Figure 26—Path constraint

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

5.7.1.2 Period constraints

The **PERIODCONSTRAINT** construct allows a path constraint value to be specified for groups of paths in a synchronous circuit. All paths in the group shall 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. The syntax for period constraint is given in Syntax 56.

> *period_constraint* ::=
> **(PERIODCONSTRAINT** *port_instance value* [*exception*] **)**
> *exception* ::=
> **(EXCEPTION** *cell_instance* { *cell_instance* } **)**

Syntax 56: Syntax for period constraint

— *port_instance* identifies the common clock signal which is the start of all constrained paths. Whereas the start of **PATHCONSTRAINT** 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 shall cause the *value* supplied in **PERIODCONSTRAINT** to be different (typically smaller) than the intended clock period at which the circuit shall operate. Since only one *value* can be supplied for all paths in this group, some data may be lost in combining many **PATHCONSTRAINT** constructs into one **PERIODCONSTRAINT**.
— *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. Therefore, the **PERIODCONSTRAINT** 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 (48):

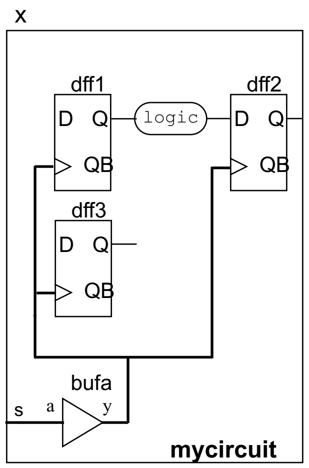


Figure 27—Period constraint

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

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

5.7.1.3 Sum constraints

The **SUM** shall represent a constraint on the sum of the delay over two or more paths in a design. The syntax for sum constraint is described in Syntax 57.

sum_constraint ::=
  (SUM *constraint_path constraint_path* { *constraint_path* } *rvalue* [*rvalue*] **)**
constraint_path ::=
    (*port_instance port_instance* **)**

Syntax 57: Syntax for sum constraint

— 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 applies to the falling.

Example (49):

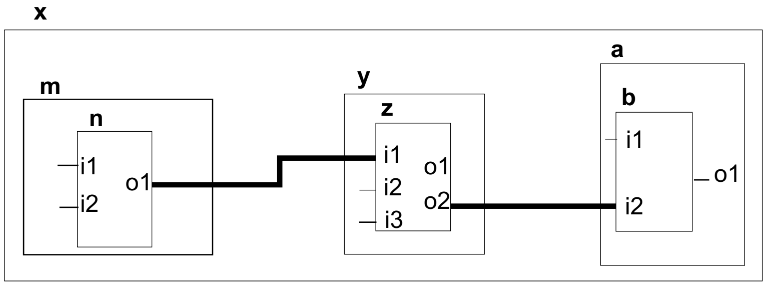


Figure 28—Sum constraint

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

Example (49) 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.

5.7.1.4 Diff constraints

DIFF shall represent a constraint on the difference in the delay over two paths in a design. The syntax for diff constraint is given in Syntax 58.

diff_constraint ::=
 (DIFF *constraint_path constraint_path value* [*value*] **)**

Syntax 58: Syntax for diff constraint

— *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 (50):

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

5.7.1.5 Skew constraints

SKEWCONSTRAINT shall represent 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. The syntax for skew constraint is shown in Syntax 59.

skew_constraint ::=
(SKEWCONSTRAINT *port_spec value* **)**

Syntax 59: Syntax for skew constraint

— *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 shall 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 (51):

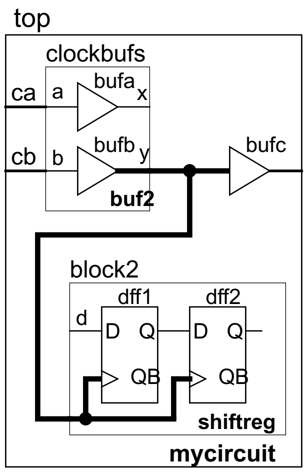


Figure 29—Skew constraint

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

In Example (51), 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** and its output is specified in a **SKEWCONSTRAINT**. The effect is to request that the arrival of the positive edge of the clock shall not deviate by more than 7.5 time units 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. Note that the driven inputs can be anywhere in the design, irrespective of the hierarchical organization.

5.7.2 Timing environment constructs

Timing environment constructs provide information about the timing environment in which the circuit shall 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. The syntax for timing environment constructs is described in Syntax 60.

```
tenv_def ::=
        arrival_env
      | departure_env
      | slack_env
      | waveform_env
arrival_env ::=
        ( ARRIVAL [ port_edge ] port_instance rvalue rvalue rvalue rvalue )
departure_env ::=
        ( DEPARTURE [ port_edge ] port_instance rvalue rvalue rvalue rvalue )
slack_env ::=
        ( SLACK port_instance rvalue rvalue rvalue rvalue [ real_number ] )
waveform_env ::=
        ( WAVEFORM port_instance real_number edge_list )
```

Syntax 60: Syntax for timing environment

Subclause 5.7.2.1 through 5.7.2.4 describes the SDF timing environment constructs.

5.7.2.1 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. The syntax for arrival time is given in Syntax 61.

```
arrival_env ::=
    ( ARRIVAL [ port_edge ] port_instance rvalue rvalue rvalue rvalue )
```

Syntax 61: Syntax for arrival time

— *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 shall be treated as the time 0 of all **WAVEFORM** constructs. Note that, to fully specify a timing environment, a **WAVEFORM** statement shall be 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 (52):

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

Example (52) specifies that rising transitions at `D[15:0]` are to be applied no sooner than 10 time units 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 time units and no later than 45 time units after the edge.

5.7.2.2 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. The syntax for departure time is given in Syntax 62.

departure_env ::=
    (DEPARTURE [*port_edge*] *port_instance rvalue rvalue rvalue rvalue* **)**

Syntax 62: Syntax for departure time

— *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 shall be treated as the time 0 of all **WAVEFORM** constructs. Note that, to fully specify a timing environment, a **WAVEFORM** statement shall be 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 (53):

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

Example (53) specifies that rising transitions at primary output `A[15:0]` are to occur no sooner than 8 time units 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 time units and no later than 34 time units after the edge.

5.7.2.3 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 shall be to eliminate negative slack and achieve an even distribution of positive slack. The syntax for slack time is given in Syntax 63.

slack_env ::=
  (SLACK *port_instance rvalue rvalue rvalue rvalue* [*real_number*] **)**

Syntax 63: Syntax for slack time

- *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 shall 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 shall have the unusual property of decreasing in value from left to right.
- *real_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 (54):

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

In Example (54), the rising and falling setup slack times are both 3, while the rising and falling hold slack times are both 7. This means that the delay of all data paths to port `macro.AOI6` could be increased by 3 time units without violating any setup requirements of devices on this net. And the delay could be decreased by 7 time units without violating any hold requirements of devices on this net.

Multiple **SLACK** constructs are allowable for the same *port_instance* and are distinct if *real_number* is different.

5.7.2.4 Waveform time

The **WAVEFORM** construct shall allow the specification of a periodic waveform that shall be applied to a circuit during its intended operation. Typically, this shall 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. The syntax for a waveform specification is described in Syntax 64.

```
waveform_env ::=
    ( WAVEFORM port_instance real_number edge_list )
edge_list ::=
          pos_pair { pos_pair }
    |     neg_pair { neg_pair }
pos_pair ::=
          ( posedge signed_real_number [ signed_real_number ] )
          ( negedge signed_real_number [ signed_real_number ] )
neg_pair ::=
          ( negedge signed_real_number [ signed_real_number ] )
          ( posedge signed_real_number [ signed_real_number ] )
```

Syntax 64: Syntax for waveform specification

— *port_instance* identifies the port in the circuit at which the waveform shall 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 shall be ignored and the specified waveform shall replace it in the analysis. The hierarchical path to this port is relative to the scope or design region identified by the cell.
— *real_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 **posedge** followed by **negedge** or **negedge** followed by **posedge**. Thus, the total number of edges in the list shall be even and edges shall 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 *signed_real_number* is supplied, then this precisely defines the transition offset. If two *signed_real_number*s are supplied, then they define an uncertainty region in which the transition shall take place. The first *signed_real_number* gives the beginning of the uncertainty region and the second *signed_real_number* gives its end. Tools using this construct with two *signed_real_number*s shall assume that a single transition of the specified direction occurs somewhere in the uncertainty region, but shall make no assumptions about exactly where. Tools unable to model this edge uncertainty shall issue a warning message and use the mean of the two *signed_real_number*s to locate the transition.

Example (55):

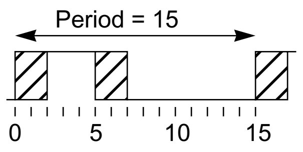


Figure 30—Specification of a waveform of period 15

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

Example (55) 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 shall place the rising edge at 1 and the falling edge at 6 and issue a warning.

Example (56):

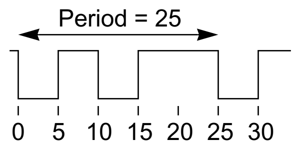


Figure 31—Specification of a waveform of period 25

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

Example (56) 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 (57):

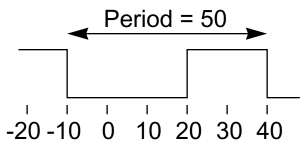


Figure 32—Specification of a waveform of period 50 with negative numbers

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

Example (57) shows that negative numbers can be used in defining a waveform.

Annex A

(normative)

Syntax of SDF

A.1 Formal syntax definition

The formal syntax of SDF is described using Backus-Naur (BNF).

Lexical elements are shown italicized. All leaf characters are shown in bold. Keywords are shown in uppercase bold (for example, **IOPATH**) for easy identification, but are case insensitive.

A.1.1 SDF delay file and header

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

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

process ::=
        (PROCESS *qstring* **)**

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

time_scale ::=
        (TIMESCALE *timescale_number timescale_unit* **)**

timescale_number ::= **1** | **10** | **100** | **1.0** | **10.0** | **100.0**

timescale_unit ::= **s** | **ms** | **us** | **ns** | **ps** | **fs**

A.1.2 Cells

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

celltype ::=
    (CELLTYPE *qstring* **)**

cell_instance ::=
    (INSTANCE [*hierarchical_identifier*] **)**
  | **(INSTANCE *)**

A.1.3 Timing specifications

timing_spec ::=
    del_spec
  | *tc_spec*
  | *lbl_spec*
  | *te_spec*

del_spec ::=
    (DELAY *deltype* { *deltype* } **)**

tc_spec ::=
    (TIMINGCHECK *tchk_def* { *tchk_def* } **)**

te_spec ::=
    (TIMINGENV *te_def* { *te_def* } **)**

lbl_spec ::=
    (LABEL *lbl_type* { *lbl_type* } **)**

deltype ::=
  | *absolute_deltype*
  | *increment_deltype*
  | *pathpulse_deltype*
  | *pathpulsepercent_deltype*

pathpulse_deltype ::=
    (PATHPULSE [*input_output_path*] *value* [*value*] **)**

pathpulsepercent_deltype ::=
    (PATHPULSEPERCENT [*input_output_path*] *value* [*value*] **)**

absolute_deltype ::=
    (ABSOLUTE *del_def* { *del_def* } **)**

increment_deltype ::=
    (INCREMENT *del_def* { *del_def* } **)**

input_output_path ::=
    port_instance port_instance

del_def ::=
iopath_def
| *cond_def*
| *condelse_def*
| *port_del*
| *interconnect_def*
| *netdelay_def*
| *device_def*

iopath_def ::=
(IOPATH *port_spec port_instance* { *retain_def* } *delval_list* **)**

retain_def ::=
(RETAIN *retval_list* **)**

cond_def ::=
(COND [*qstring*] *conditional_port_expr iopath_def* **)**

condelse_def ::=
(CONDELSE *iopath_def* **)**

port_def ::=
(PORT *port_instance delval_list* **)**

interconnect_def ::=
(INTERCONNECT *port_instance port_instance delval_list* **)**

netdelay_def ::=
(NETDELAY *net_spec delval_list* **)**

device_def ::=
(DEVICE [*port_instance*] *delval_list* **)**

tchk_def ::=
setup_timing_check
| *hold_timing_check*
| *setuphold_timing_check*
| *recovery_timing_check*
| *removal_timing_check*
| *recrem_timing_check*
| *skew_timing_check*
| *bidirectskew_timing_check*
| *width_timing_check*
| *period_timing_check*
| *nochange_timing_check*

setup_timing_check ::=
(SETUP *port_tchk port_tchk value* **)**

hold_timing_check ::=
(HOLD *port_tchk port_tchk value* **)**

setuphold_timing_check ::=
(SETUPHOLD *port_tchk port_tchk rvalue rvalue* **)**
| **(SETUPHOLD** *port_spec port_spec rvalue rvalue* [*scond*] [*ccond*] **)**

recovery_timing_check ::=
(RECOVERY *port_tchk port_tchk value* **)**

removal_timing_check ::=
(REMOVAL *port_tchk port_tchk value* **)**

recrem_timing_check ::=
    (RECREM *port_tchk port_tchk rvalue rvalue* **)**
  | **(RECREM** *port_spec port_spec rvalue rvalue* [*scond*] [*ccond*] **)**

skew_timing_check ::=
    (SKEW *port_tchk port_tchk rvalue* **)**

bidirectskew_timing_check ::=
    (BIDIRECTSKEW *port_tchk port_tchk value value* **)**

width_timing_check ::=
    (WIDTH *port_tchk value* **)**

period_timing_check ::=
    (PERIOD *port_tchk value* **)**

nochange_timing_check ::=
    (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* **)**

te_def ::=
    cns_def
  | *tenv_def*

cns_def ::=
    path_constraint
  | *period_constraint*
  | *sum_constraint*
  | *diff_constraint*
  | *skew_constraint*

path_constraint ::=
    (PATHCONSTRAINT [*name*] *port_instance port_instance* { *port_instance* }
        rvalue rvalue **)**

period_constraint ::=
    (PERIODCONSTRAINT *port_instance value* [*exception*] **)**

sum_constraint ::=
    (SUM *constraint_path constraint_path* { *constraint_path* } *rvalue* [*rvalue*] **)**

diff_constraint ::=
    (DIFF *constraint_path constraint_path value* [*value*] **)**

skew_constraint ::=
    (SKEWCONSTRAINT *port_spec value* **)**

exception ::=
    (EXCEPTION *cell_instance* { *cell_instance* } **)**

name ::=
    (NAME [*qstring*] **)**

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

tenv_def ::=
    arrival_env
  | *departure_env*
  | *slack_env*
  | *waveform_env*

arrival_env ::=
    (ARRIVAL [*port_edge*] *port_instance rvalue rvalue rvalue rvalue* **)**

departure_env ::=
    (DEPARTURE [*port_edge*] *port_instance rvalue rvalue rvalue rvalue* **)**

slack_env ::=
    (SLACK *port_instance rvalue rvalue rvalue rvalue* [*real_number*] **)**

waveform_env ::=
    (WAVEFORM *port_instance real_number edge_list* **)**

edge_list ::=
    pos_pair { *pos_pair* }
  | *neg_pair* { *neg_pair* }

pos_pair ::=
    (posedge *signed_real_number* [*signed_real_number*] **)**
    (negedge *signed_real_number* [*signed_real_number*] **)**

neg_pair ::=
    (negedge *signed_real_number* [*signed_real_number*] **)**
    (posedge *signed_real_number* [*signed_real_number*] **)**

lbl_type ::=
    (INCREMENT *lbl_def* { *lbl_def* } **)**
  | **(ABSOLUTE** *lbl_def* { *lbl_def* } **)**

lbl_def ::=
    (*identifier delval_list* **)**

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
  | *hierarchical_identifier hchar port*

port ::=
  scalar_port
| *bus_port*

scalar_port ::=
  hierarchical_identifier
| *hierarchical_identifier* **[** *integer* **]**

bus_port ::=
  hierarchical_identifier **[** *integer* **:** *integer* **]**

net_spec ::=
  port_instance
| *net_instance*

net_instance ::=
  net
| *hierarchical_identifier hier_divider_char net*

net ::=
  scalar_net
| *bus_net*

scalar_net ::=
  hierarchical_identifier
| *hierarchical_identifier* **[** *integer* **]**

bus_net ::=
  hierarchical_identifier **[** *integer* **:** *integer* **]**

A.1.4 Data values

value ::=
  ([*real_number*] **)**
| **(** [*triple*] **)**

triple ::=
  real_number **:** [*real_number*] **:** [*real_number*]
| [*real_number*] **:** *real_number* **:** [*real_number*]
| [*real_number*] **:** [*real_number*] **:** *real_number*

rvalue ::=
  ([*signed_real_number*] **)**
| **(** [*rtriple*] **)**

rtriple ::=
  signed_real_number **:** [*signed_real_number*] **:** [*signed_real_number*]
| [*signed_real_number*] **:** *signed_real_number* **:** [*signed_real_number*]
| [*signed_real_number*] **:** [*signed_real_number*] **:** *signed_real_number*

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

```
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 ]

retval_list ::=
        delval
    |   delval delval
    |   delval delval delval
```

A.1.5 Conditions for path delays

```
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 ? simple_expression : simple_expression
    |   { simple_expression [ concat_expression ] }
    |   { simple_expression { simple_expression [ concat_expression ] } }

concat_expression ::=
        , simple_expression
```

A.1.6 Conditions for timing checks

```
timing_check_condition ::=
        scalar_node
    |   inversion_operator scalar_node
    |   scalar_node equality_operator scalar_constant

scalar_node ::=
        scalar_port
    |   scalar_net

scalar_net ::=
        hierarchical_identifier
```

A.1.7 Fundamental lexical elements

White space is normally permitted between lexical tokens, but not within the definitions in A.1.7.

```
identifier ::= character { character }

hierarchical_identifier ::=identifier { hchar identifier }

qstring ::= " { any_character } "
```

integer ::= *decimal_digit* { *decimal_digit* }

real_number ::=
    integer
| *integer* [**.** *integer*]
| *integer* [**.** *integer*] **e** [*sign*] *integer*

signed_real_number ::= [*sign*] *real_number*

sign ::= **+** | **-**

hchar ::= **.** | **/**

character ::=
    alphanumeric
| *escaped_character*

escaped_character ::=
    \ *character*
| \ *special_character*
| **\\"**

any_character ::=
    character
| *special_character*
| **\\"**

decimal_digit ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

alphanumeric ::=
    a | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z**
| **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M** | **N** | **O** | **P** | **Q** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z**
| **_** | **$**
| *decimal_digit*

special_character ::=
    ! | **#** | **%** | **&** | **«** | **(** | **)** | ***** | **+** | **,** | **-** | **.** | **/** | **:** | **;** | **<** | **=** | **>** | **?** | **@** | **[** | \ | **]** | **^** | **‘** | **{** | **|** | **}** | **~**

A.1.8 Constants for expressions

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

A.1.9 Operators for expressions

```
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
```

A.1.10 Precedence rules of SDF operators

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

Annex B

(informative)

SDF file examples

B.1 SDF file example 1

This SDF file example is based on the schematic shown in Figure B.1.

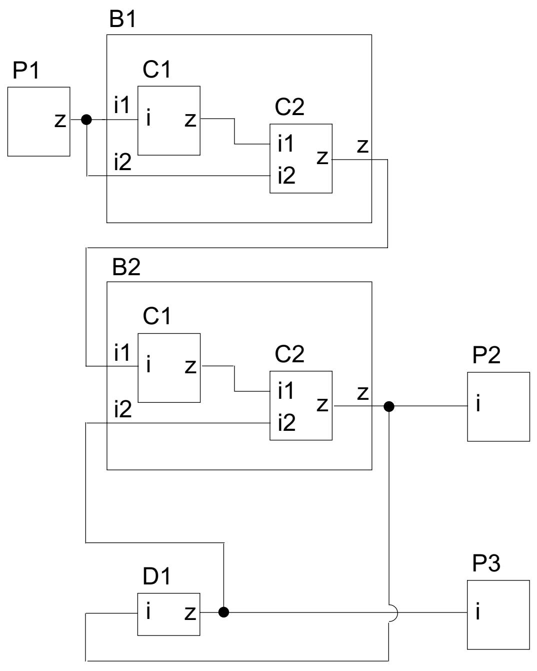


Figure B.1—SDF example schematic

```
(DELAYFILE
   (SDFVERSION "4.0")
   (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) )
         )
      )
   )
)
```

B.2 SDF file example 2

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

```
(DELAYFILE
   (SDFVERSION "4.0")
   (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) )
      )
   )
)
```

B.3 SDF file example 3

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

```
(DELAYFILE
   (SDFVERSION "4.0")
   (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) ) )
          )
       )
   )
)
```

B.4 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 following example, I2 is an instance and H01 is a PIN (port) on that instance.

```
(DELAYFILE
   (SDFVERSION "4.0")
   (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) )
      )
   )
)
```