IEEE Standard for VITAL Application-Specific Integrated Circuit (ASIC) Modeling Specification

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society

Approved December 12, 1995

IEEE Standards Board

Abstract: The VITAL (VHDL Initiative Towards ASIC Libraries) ASIC Modeling Specification is defined. It creates a methodology that promotes the development of highly accurate, efficient simulation models for ASIC (Application-Specific Integrated Circuit) components in VHDL.
Keywords: ASIC, computer, computer languages, constraints, delay calculation, HDL, modeling, SDF, timing, Verilog®, VHDL

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street, New York, NY 10017-2394, USA

ISBN 1-55937-691-0

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason IEEE and the members of its technical committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

Note: Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying all patents for which a license may be required by an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

Introduction

[This introduction is not a part of IEEE Std 1076.4-1995, IEEE Standard for VITAL Application-Specific Integrated Circuit (ASIC) Modeling Specification.]

The objective of VITAL (VHDL Initiative Towards ASIC Libraries) was to accelerate the development of sign-off quality ASIC macrocell simulation libraries written in the VHSIC Hardware Description Language (VHDL) by leveraging existing methodologies of model development.

The VITAL effort germinated from ideas generated at the VHDL International Users' Forum in May 1992. Further discussions revealed that the biggest impediment to VHDL design was the lack of ASIC libraries, and that the biggest impediment to ASIC library development was the lack of a uniform, efficient method for handling timing in VHDL. Since this problem had already been solved for other languages, it was clear that a solution in VHDL was possible and that an effective way to arrive at this solution was to leverage existing technology. Leveraging existing tools and environments serves as a catalyst for the rapid deployment of ASIC libraries.

VITAL obtains its leverage from

a) The Standard Delay Format (SDF), in addition to

b) Certain contributed elements of the Std_Timing package provided by William Billowitch and specialized timing and behavioral techniques provided by Ray Ryan, and

c) The existence of numerous ASIC libraries and tools developed using SDF timing implementations

VITAL used many ideas about the primitives and timing models described in the Verilog® language. In particular, Verilog's support for representing truth/state tables and its mechanism for performing pin-to-pin delay selection were found to be highly useful.

The VITAL organization was an informal consortium of interested companies in the electronics industry who shared the goals of accelerating the availability of VHDL ASIC libraries. They represented the three components of the ASIC design triangle: ASIC vendors, EDA vendors, and end users of ASIC components. More than 60 such companies worldwide joined the consortium. This group was led by a steering committee, which was responsible for the development of the VITAL technical specifications as well as the promotion and dissemination of information about this work. The members of the steering committee were: Steven Schulz, William Billowitch, Ray Ryan, Oz Levia, Victor Berman, Victor Martin, Ravi Kumar, Sanjay Nayak, Tom Senna, and Herman Van Beek.

The VITAL steering committee transferred this work to the IEEE P1076.4 Working Group for consideration as the basis of a standard for ASIC modeling in VHDL. This standard is the result of the efforts of the working group in refining that baseline document.

The IEEE P1076.4 Working Group has a membership of over 300 interested people who have made significant contributions to this work through their participation in technical meetings; their review of technical data both in print and through electronic media; and their votes, which guided and finally approved the content of the draft standard.

The technical direction of the working group as well as the day-to-day activities of issue analysis and drafting of proposed wordings for the standard were the responsibility of the IEEE P1076.4 TAG (Technical Action Group). This group consisted of Ravi Kumar, Sanjay Nayak, Dennis Brophy, Ray Ryan, John Busco, Tim Ayres, Bill Paulsen, and Kathy McKinley, and the group was chaired by Oz Levia. Without the dedication and hard work of this group, it would not have been possible to complete this work.

This standard is the result of numerous discussions with ASIC vendors, CAE tool vendors, and ASIC designers

to determine the requirements for effective design and fabrication of ASICs using VHDL. The highest priority issues identified by this group were

— Timing accuracy

— Model maintainability

— Simulation performance

Some basic guiding principles followed during the entire specification development process were

— To describe all functionality and timing semantics of the model entirely within the VHDL model and the associated VITAL packages.

— To provide a set of modeling rules (Level 1) that constrain the use of VHDL to a point that is amenable for simulator optimizations, and at the same time to provide enough flexibility to support most existing modeling scenarios.

— To have all timing calculations (load dependent or environmentally dependent) performed outside of the VITAL model. The VITAL model would get these timing values solely as actual values to the generic parameter list of the model or via SDF direct import.

The following persons were members of the IEEE P1076.4 Working Group:

Victor Berman, *Chair*

Kathy McKinley, *Technical Editor*

Suresh Agarwal
Dave Agnew
Dave Allenbaugh
Brien Anderson
Cliff Anderson
Jan Anderson
Robert E. Anderson
Ronen Arad
Libby Aston
Naveena N. Aswadhati
Jeffrey M. Aubert
Larry M. Augustin
Bulent Ay
Tim Ayres
Stephen A. Bailey
Mikhail A. Baklashov
Ekambaram Balaji
Bruce Bandali
Sudarshan Banerjee
Peter Barck
Daniel Barclay
Karen Bartleson
Mark Basten
Michael A. Beaver
David Belz
Leon Benders
Ad J. W. M. ten Berg
Jean-Michel Berge
Werner Bergmann
David Bernstein
Christian Berthet
Jayaram Bhasker
Scott Bilik
William D. Billowitch
John Biro
[illegible]
Robert Bloor
Michael Bohm
Frederique Bouchard
Bruce Bourbon
Elaine Boyd
Jean Pierre Braunt
Christopher Brock
Dennis Brophy
John A. Busco
Jean-Paul Caisso
J. Scott Calhoun
Brian Caslis
Shir-Shen Chang
Thomas Chao
Clive Charlwood
Jean Marc Chateau
Glenn J. Childers
Nitin Chowdhary
Debashis Roy Chowdhury
Ron Christopher
Carl Cleaver
David Coelho
John Colley
Tedd Corman
Joe Costello
Terry Coston
Ronnie Craig
Ronan Cullen
Hal Daseking
Jose De Castro
Aart deGeus
Jean Desuche
Baruch Deutsch
Allen Dewey
Matt Dodd
Kevin Donnelly
Mike Duclos
Douglas D. Dunlop
Hisakazu Edamatsu
Steve Elliott
Stephan Eriksson
David Evans
Edward N. Evans
John A. Evans
William Fazakerly
Francine Ferguson
Len Finegold
Geoffrey Frank
Roberta E. Fulton
Sanjay Goswami
Ravender Goyal
Brian Griffin
Gary Griffin

Bruce Grugett
Andrew Guyler
Werner Hack
Klaus ten Hagen
Stuart Hamilton
James P. Hanna
Alain Hanover
Susan Hardenbrook
David Hardman
Randolph E. Harr
Amirhooshang Hashemi
Greg Haynes
Carl E. Hein
Mitch Heins
Shankar Hemmady
Moises Hernandez
Elhanan Herzog
Kiyohiro Higashijima
John Hillawi
Fred Hinchcliffe
John Hines
Helmut Hissen
Will Hobbs
Andreas Hohl
Yu-I Hsieh
Jason Hsiung
Anling Hsu
Matt Hsu
Pong Hsu
May Huang
David Hubbard
Christophe Hui-Bon-Hoa
Erik Huyskens
Kazuhiko Iijima
Stephen A. Ives
Jacob Jacobson
Antonie C. S. de Jager
Mahendra Jain
Sunil K. Jain
Colin Jitlal
Don Johansen
Howard Johnson
Harvey Jones
Mark Jones
Masaru Kakimoto
Ken Kappeler
Jake Karrfalt
Eugene Ko
Satoshi Kojima
Rama Kowsalya
Stanley J. Krolikoski
Paul Krueger
Krishna Kumar
Ravi Kumar
C. Lakshmikantam
Sylvie Lasserre
Marc Laurent
Manu Lauria
Oz Levia
Serge Maginot
Arthur Magnan
Sanjay Malpani
John Mancini
Maq Mannan
F. Erich Marschner
Victor M. Martin
Larry Melling
Paul J. Menchini
Alex Miczo
Dwight Miller
Nikhil Modi
Kent Moffat
Sidhartha Mohanty
Gabe Moretti
Rick Munden
Jayant L. Nagda
Zain Navabi
Sanjay Nayak
Gordon L. Newell
John O'Brien
William R. Paulsen
Zamir Paz
Venu Pemmaraju
Gabriele Pulini
Viswanathan Ramakrishnan
Paul Ramondetta
Prakash Reddy
Praveen Reddy
Eugen Roehm
Adam Rosenberg
Arnob Roy
Susan Runowicz-Smith
Ray Ryan
Yvonne Ryan
Hideki Sagayama
Kyohei Sakajiri
Manohar Sambandam
Johan Sandstrom
Marc-Alain Santerre
Souvanik Sarkar
Paul Sathya
Sharada Satrasala
Larry F. Saunders
Quentin Schmierer
Joel Schoen
Steven E. Schulz
David Sellers
Tom Senna
Moe Shahdad
Steve Sherman
Venk Shukla
Ken Simone
Peter Sinander
Rajvinder Singh
John Sissler
Joseph Skudlarek
Balachandran Sreekandath
John Stickley
Eugena Talvola
Tina Tran
Cary Ussery
Alain Vachoux
Vijay Vaidyanathan
Tom VandenBerge
Eugenio Villar
Walter Vines
Jefferry Vo
Russ Vreeland
Ron Werner
Paul Williams
John C. Willis
Jim Wilmore
Scott Winick
Alex Zamfirescu

The following persons were on the balloting committee:

William J. Abboud
Mostapha Aboulhamid
Robert E. Anderson
Stephen A. Bailey
Pete Bakowski
Jean-Michel Berge
Victor Berman
Jayaram Bhasker
William D. Billowitch
Dennis B. Brophy
John A. Busco
J. Scott Calhoun
Raul Camposano
Harold W. Carter
Shir-Shen Chang
Thomas Chao
Chin-Fu Chen
Mojy C. Chian
Pradeep Chilka
David Coelho
John Colley
Alan Coppola
Tedd Corman
W. Terry Coston
Joanne DeGroat
Antonie C. deJager
Allen Dewey
Michael A. Dukes
Douglas D. Dunlop
Ted Elkind
Edward N. Evans
William Fazakerly
Peter Flake
Jacques P. Flandrois
Rita A. Glover
Rich Goldman
Brian Griffin
Richard Grisel
Steve Grout
Andrew Guyler
William A. Hanna
James P. Hanna
Randolph E. Harr
John Hillawi
Robert G. Hillman
Yu-I Hsieh
Yee-Wing Hsieh
May Huang
Christophe Hui-Bon-Hoa
Sylvie Hurat
Masaharu Imai
Ann Irza
Mitsuaki Ishikawa
Stephen A. Ives
Navneet Kumar Jain
Choon B. Kim
Stanley J. Krolikoski
Jean Lebrun
Oz Levia
Shirley Lu
Rajeev Madhavan
Serge Maginot
F. Erich Marschner
Victor M. Martin
Paul J. Menchini
Gerald T. Michael
John T. Montague
Gabe Moretti
Jayant L. Nagda
Sanjay Nayak
Kevin O'Brien
Curtis Parks
William R. Paulsen
Mauro Pipponzi
Gary S. Porter
Adam Postula
Jan Pukite
Rami Rahim
Hemant G. Rotithor
Larry F. Saunders
Quentin Schmierer
Steven E. Schulz
Francesco Sforza
Ravi Shankar
Micahel D. Sharp
Raj Singh
Djahida Smati
Geoffrey John Smith
J. Dennis Soderberg
Alec G. Stanculescu
Balsha R. Stanisic
Michael F. Sullivan
Charles Swart
Peter Ting
Cary Ussery
E. Vandris
Ranganadha R. Vemuri
Venkat V. Venkataraman
Eugenio Villar
Ronald Waxman
Ron Werner
Alan Whittaker
John C. Willis
Mark Zwolinski

When the IEEE Standards Board approved this standard on December 12, 1995, it had the following membership:

E. G. "Al" Kiener, *Chair*

Donald C. Loughry, *Vice Chair*

Andrew G. Salem, *Secretary*

Gilles A. Baril
Clyde R. Camp
Joseph A. Cannatelli
Stephen L. Diamond
Harold E. Epstein
Donald C. Fleckenstein
Jay Forster*
Donald N. Heirman
Richard J. Holleman
Jim Isaak
Ben C. Johnson
Sonny Kasturi
Lorraine C. Kevra
Ivor N. Knight
Joseph L. Koepfinger*
D. N. "Jim" Logothetis
L. Bruce McClung
Marco W. Migliaro
Mary Lou Padgett
John W. Pope
Arthur K. Reilly
Gary S. Robinson
Ingo Rüsch
Chee Kiow Tan
Leonard L. Tripp
Howard L. Wolfman

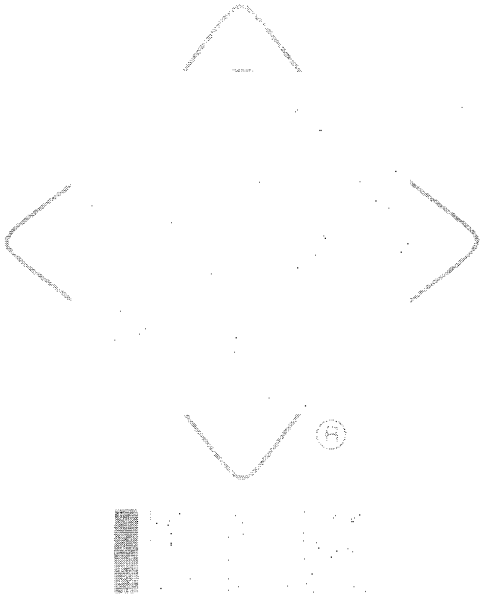
*Member Emeritus

Also included are the following nonvoting IEEE Standards Board liaisons:

Satish K. Aggarwal
Steve Sharkey
Robert E. Hebner
Chester C. Taylor

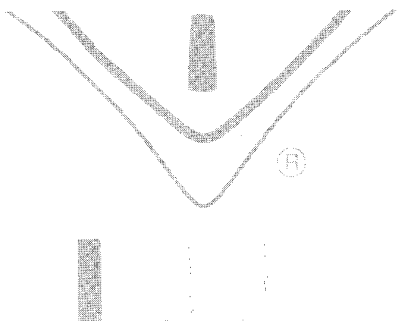
Mary Lynne Nielsen
IEEE Standards Project Editor

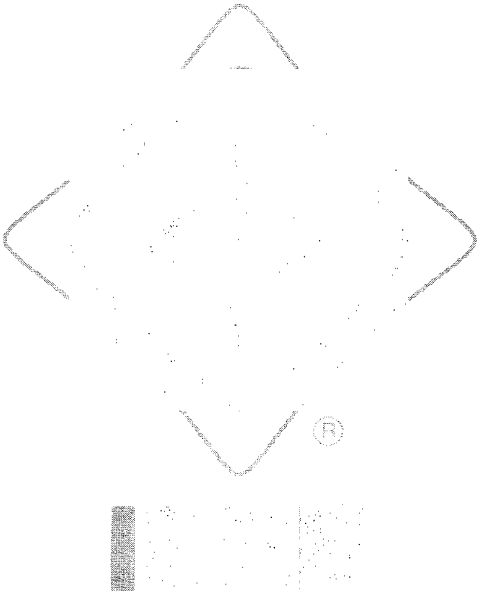
Verilog is a registered trademark of Cadence Design Systems, Inc.



Contents

CLAUSE PAGE





IEEE Standard for VITAL Application-Specific Integrated Circuit (ASIC) Modeling Specification

1. Overview

This clause describes the purpose and organization of this standard.

1.1 Intent and scope of this standard

The intent of this standard is to define accurately the standard VITAL ASIC modeling specification. The primary audience for this document are the implementors of tools supporting the standard and ASIC modelers.

1.2 Structure and terminology of this standard

This standard is organized into clauses, each of which focuses on some particular area of the definition of the specification. Each page of the formal definition contains ruler-style line numbers in the left margin. Within each clause, individual constructs or concepts are discussed in each subclause.

Each subclause describing a specific construct or concept begins with an introductory paragraph. If applicable, the syntax of the construct is then described using one or more grammatical productions. A set of paragraphs describing in narrative form the information and rules related to the construct or concept then follows. Finally, each subclause may end with examples, figures, and notes.

1.3 Syntactic description

The form of a VITAL-compliant VHDL description is described by means of a context-free syntax, using a simple variant of the Backus-Naur form; in particular:

a) Lowercased words, some containing embedded underlines, are used to denote syntactic categories, for example:

VITAL_process_statement

Whenever the name of a syntactic category is used, apart from the syntax rules themselves, spaces take the place of underlines (thus, "VITAL process statement" would appear in the narrative description when referring to the above syntactic category).

b) Boldface words are used to denote reserved words, for example:

process

Reserved words shall be used only in those places indicated by the syntax.

c) A *production* consists of a *left-hand side*, the symbol "::=" (which is read as "can be replaced by"), and a *right-hand side*. The left-hand side of a production is always a syntactic category; the right-hand side is a replacement rule.

The meaning of a production is a textual-replacement rule: any occurrence of the left-hand side may be replaced by an instance of the right-hand side.

d) A vertical bar separates alternative items on the right-hand side of a production unless it occurs immediately after an opening brace, in which case it stands for itself.

e) Square brackets enclose optional items on the right-hand side of a production.

f) Braces enclose a repeated item or items on the right-hand side of a production. The items may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule.

g) If the name of any syntactic category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example, *unrestricted_variable*_name is syntactically equivalent to name alone.

h) The term simple_name is used for any occurrence of an identifier that already denotes some declared entity.

i) A syntactic category for which no replacement rule is specified is assumed to correspond to the VHDL syntactic category of the same name. In this case the appropriate replacement rule can be found in IEEE Std 1076-1987.[1]

j) A syntactic category beginning with the unitalicized prefix "VITAL_" represents a subset of a VHDL syntactic category.

k) A word shown in all uppercase letters can represent a reserved word in VHDL.

l) SDF constructs are represented in the following manner:

1) An italicized lowercase identifier represents an SDF syntax construct.

2) The definition of a syntax construct is indicated by the symbol ::=, and alternative definitions are separated by the symbol ||=.

3) Keywords appear in boldface capital letters.

4) Uppercase identifiers represent variable symbols.

5) The form "item?" represents an optional item.

[1]Information about references can be found in clause 2.

6) The form "item*" represents zero or more occurrences of the item.

7) The form "item+" indicates one or more occurrences of the item.

1.4 Semantic description

The meaning of a particular construct or concept and any related restrictions are described with a set of narrative rules immediately following any syntactic productions in the subclause. In these rules, an italicized term indicates the definition of that term, and an identifier appearing in Helvetica font refers to a definition in one of the VHDL or VITAL standard packages or in a VHDL model description. An identifier beginning with the prefix "VITAL" corresponds to a definition in a VITAL standard package.

Use of the words "is" or "shall" in such a narrative indicates mandatory weight. A noncompliant practice may be described as *erroneous* or as an *error*. These terms are used in these semantic descriptions with the following meaning:

erroneous—the condition described represents a noncompliant modeling practice; however, implementations are not required to detect and report this condition. Conditions are deemed erroneous only when it is either very difficult or impossible in general to detect the condition during the processing of a model.

error—the condition described represents a noncompliant modeling practice; implementations are required to detect the condition and report an error to the user of the tool.

1.5 Front matter, examples, figures, notes, and annexes

Prior to this clause are several pieces of introductory material; following the final clause are some annexes and an index. The front matter, annexes, and index serve to orient and otherwise aid the user of this standard but are not part of the definition of this standard.

Some subclauses of this standard contain examples, figures, and notes; with the exception of figures, these parts always appear at the end of a subclause. Examples are meant to illustrate the possible forms of the construct described. Figures are meant to illustrate the relationship between various constructs or concepts. Notes are meant to emphasize consequences of the rules described in the clause or elsewhere. In order to distinguish notes from the other narrative portions of the definition, notes are set as enumerated paragraphs in a font smaller than the rest of the text. Examples, figures, and notes are not part of the definition of the specification.

2. References

This standard shall be used in conjunction with the following publications. Bibliographic references may be found in annex C. Citations of the form "[B1]" refer to items listed in annex C, not to items listed in this clause.

IEEE Std 1076-1987, IEEE Standard VHDL Language Reference Manual. [2]

IEEE Std 1164-1993, IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164).[3]

Standard Delay Format Specification, Version 2.1.[4]

[2]IEEE Std 1076-1987 has been superceded by IEEE Std 1076-1993 (see Annex C for bibliographic information on the latest version of this standard). IEEE Std 1076-1987 is no longer in print; however, it is available archivally from Global Engineering, 15 Inverness Way East, Englewood, CO 80112-5704, USA.

[3]IEEE publications are available from the Institute of of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA.

[4]OVI publications are available from Open Verilog International (OVI), 15466 Los Gatos Blvd, Suite 109-071, Los Gatos, CA 95032.

3. Basic elements of the VITAL ASIC modeling specification

This standard defines a modeling style for the purpose of facilitating the development and acceleration of sign-off quality ASIC macrocell simulation libraries written in VHDL.

This standard is an application of the VHSIC Hardware Description Language (VHDL), described in IEEE Std 1076-1987 and IEEE Std 1076-1993 [B1]. This standard uses the term *VHDL* to refer to the VHSIC Hardware Description Language. In no case shall the modeling rules introduced by this standard be in conflict with IEEE Std 1076-1987.

This standard relies on the IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_Logic_1164), described in IEEE Std 1164-1993, for its basic logic representation. Throughout this standard, the term *standard logic* refers to the Std_Logic_1164 package or to an item declared in the Std_Logic_1164 package.

This standard relies on the Standard Delay Format (SDF), as defined by the OVI Standard Delay Format Specification Version 2.1, as a standard external timing data representation. Throughout this standard, the term *SDF* refers to this particular version of the delay format.

This standard consists of three basic elements: the formal definition of a VITAL-compliant VHDL model, a set of VHDL packages for providing standard timing support and standard functionality support, and a semantic specification describing a standard mechanism for insertion of timing data into a VHDL model.

3.1 VITAL modeling levels and compliance

A VITAL ASIC cell is represented by a VHDL design entity. This standard defines the characteristics of a VITAL design entity in terms of the VHDL descriptions of the entity and architecture, and in terms of the associated model which is the result of the elaboration of those VHDL descriptions.

This standard defines two modeling levels; these levels are called *VITAL Level 0* and *VITAL Level 1*. Each modeling level is defined by a set of modeling rules. The VITAL Level 0 specification forms a proper subset of the VITAL Level 1 specification.

A model is said to adhere to the rules in a particular specification only if both the model and its VHDL description satisfy all of the requirements of the specification. Furthermore, if such a model makes use of an item described in a configuration declaration or a package other than a VHDL or VITAL standard package, then the external item shall satisfy the requirements of the specification, as though the item appeared in the VHDL description of the design entity itself.

The VITAL Level 0 specification defines a set of standard modeling rules that facilitate the portability and interoperability of ASIC models, including the specification of timing information. A model that adheres to the rules in the VITAL Level 0 specification is said to be a *VITAL Level 0 model*. The VITAL Level 0 modeling specification is described in clause 4.

The VITAL Level 1 specification defines a usage model for constructing complete cell models in a manner that

facilitates optimization of the execution of the models. A model that adheres to the rules in both the VITAL Level 0 model interface specification and the VITAL Level 1 model architecture specification is said to be a *VITAL Level 1 model.* The VITAL Level 1 modeling rules are defined in clause 6.

A model that is a VITAL Level 0 model or a VITAL Level 1 model is said to be *VITAL compliant.* A VITAL-compliant model description contains an attribute specification representing the highest level of compliance intended by the enclosing entity or architecture. Descriptions of these attribute specifications may be found in 4.1 and 6.1.

NOTES

1—A VITAL Level 1 model is by definition a VITAL Level 0 model as well (but not vice versa).

2—The rules outlined in the VITAL Level 0 and VITAL Level 1 specifications apply to model descriptions, not to the VITAL standard packages themselves.

3—A VITAL-compliant tool is assumed to enforce the definition of all applicable rules in accordance with the definitions of the terms IS, SHALL, ERROR, and ERRONEOUS. In addition, a compliant tool is expected to accept and execute correctly a VITAL-compliant model, and to identify and reject models that are not compliant. A VITAL-compliant tool is also expected to support fully the processes described in the specification, including SDF backannotation and negative time sequential constraint transformation.

3.2 VITAL standard packages

This standard defines two standard packages for use in specifying the timing and functionality of a model: VITAL_Timing and VITAL_Primitives. The text of these packages may be found in clause 10.

The VITAL_Timing package defines data types and subprograms to support development of macrocell timing models. Included in this package are routines for delay selection, output scheduling, and timing violation checking and reporting.

The VITAL_Primitives package defines a set of commonly used combinatorial primitives and general-purpose truth and state tables. The primitives are provided in both function and concurrent procedure form to support both behavioral and structural modeling styles.

3.3 VITAL specification for timing data insertion

This standard defines certain semantics that are assumed by a VITAL-compliant model and shall be implemented by a tool processing or simulating VITAL-compliant models that rely on these semantics. These semantics concern the specification and processing of timing data in a VHDL model. They cover SDF mapping, backannotation, and negative constraint processing.

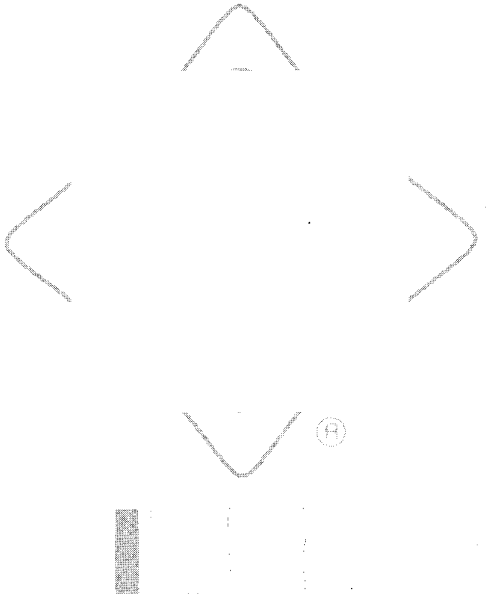
The timing data for a VITAL-compliant model may be specified in Standard Delay Format (SDF). The VITAL SDF map is a mapping specification that defines the translation between SDF constructs and the corresponding generics in VITAL-compliant models. The mapping specification may be used by tools to insert timing information into a VHDL model, either by generating an appropriate configuration declaration or by performing backannotation through direct SDF import. The VITAL SDF map is defined in 5.2.

This standard introduces two new simulation phases for designs using VITAL models: the backannotation phase and the negative constraint calculation phase. These phases occur after VHDL elaboration but before initialization.

The backannotation specification defines a backannotation phase of simulation and a mechanism for directly

annotating generics with appropriate timing values from SDF (see 5.1.1). The specification also defines the correct state of the timing generics of a model at the end of the backannotation phase.

The negative constraint calculation specification describes a methodology for modeling negative timing constraints in VHDL (see 8.2). It defines a negative constraint calculation phase of simulation and an algorithm for computing and adjusting signal delays, which together transform the negative delays into nonnegatives for the purpose of simulation.



4. The VITAL Level 0 specification

The VITAL Level 0 specification is a set of modeling rules that promotes the portability and interoperability of model descriptions by outlining general standards for VHDL language usage, restricting the form and semantic content of VITAL Level 0 design entity descriptions, and standardizing the specification and processing of timing information. General VITAL Level 0 modeling rules are defined in this clause, and those relating to the modeling of negative timing constraints are defined in 8.2.1.

4.1 The VITAL_Level0 attribute

A VITAL Level 0 entity or architecture is identified by its decoration with the VITAL_Level0 attribute, which indicates an intention to adhere to the VITAL Level 0 specification.

VITAL_Level0_attribute_specification ::= attribute_specification

A VITAL Level 0 entity or architecture shall contain a specification of the VITAL_Level0 attribute corresponding to the declaration of that attribute in the package VITAL_Timing. The entity specification of the decorating attribute specification shall be such that the enclosing entity or architecture inherits the VITAL_Level0 attribute. The expression in the VITAL_Level0 attribute specification shall be the Boolean literal True.

NOTE—Because the required attribute specification representing VITAL compliance indicates the highest level of compliance (see 3.1), a VITAL Level 1 architecture, which is also by definition a VITAL Level 0 architecture, contains a VITAL_Level1 attribute specification (see 6.1) rather than a VITAL_Level0 attribute specification. The above rules apply to architectures that are only VITAL Level 0.

Example:

```
attribute VITAL_Level0 of VitalCompliantEntity : entity is True;
```

4.2 General usage rules

A VITAL Level 0 model shall adhere to general usage rules that address portability and interoperability.

Rules that reference an item declared in a VHDL standard package, a VITAL standard package, or the Std_Logic_1164 package require the use of that particular item. A model description shall not use VHDL scope or visibility rules to declare or use an alternative item with the same name in the place of the item declared in one of these packages.

4.2.1 Standard VHDL usage

A VITAL Level 0 model is restricted to the use of IEEE Std 1076-1987 features that are portable, as defined by IEEE Std 1076/INT-1991 [B2]. Use of foreign architecture bodies or package bodies is prohibited. In addition, a VITAL-compliant model shall not use any features of IEEE 1076-1987 that have been removed or modified in the language revision described in IEEE Std 1076-1993 [B1], nor shall it use any keywords or features introduced in IEEE Std 1076-1993 [B1].

It is erroneous for a VITAL model to make use of vendor-supplied attributes or other informative VHDL constructs, such as meta-comments or directives, in a manner that affects the function or timing characteristics of the model.

4.2.2 Organization of VITAL-compliant descriptions

The VHDL design entity representing a VITAL ASIC cell is described by a pair of design units that reside in one or more VHDL design files. This standard imposes no special requirements on the placement of VITAL-compliant descriptions within design files, which may contain a mixture of compliant and noncompliant descriptions.

```
VITAL_design_file ::=
    VITAL_design_unit { VITAL_design_unit }

VITAL_design_unit ::=
      context_clause library_unit
    | context_clause VITAL_library_unit

VITAL_library_unit ::=
      VITAL_Level_0_entity_declaration
    | VITAL_Level_0_architecture_body
    | VITAL_Level_1_architecture_body
```

4.3 The VITAL Level 0 entity interface

A VITAL Level 0 entity declaration defines an interface between a VITAL-compliant model and its environment.

```
VITAL_Level_0_entity_declaration ::=
    entity identifier is
        VITAL_entity_header
        VITAL_entity_declarative_part
    end [ entity_simple_name ] ;

VITAL_entity_header ::=
    [ VITAL_entity_generic_clause ]
    [ VITAL_entity_port_clause ]

VITAL_entity_generic_clause ::=
    generic ( VITAL_entity_interface_list ) ;

VITAL_entity_port_clause ::=
    port ( VITAL_entity_interface_list ) ;

VITAL_entity_interface_list ::=
    VITAL_entity_interface_declaration { ; VITAL_entity_interface_declaration }

VITAL_entity_interface_declaration ::=
      interface_constant_declaration
    | VITAL_timing_generic_declaration
    | VITAL_control_generic_declaration
    | VITAL_entity_port_declaration

VITAL_entity_declarative_part ::= VITAL_Level0_attribute_specification
```

The form of this interface strictly limits the use of declarations and statements. The only form of declaration allowed in the entity declarative part is the specification of the VITAL_Level0 attribute. No statements are allowed in the entity statement part.

4.3.1 Ports

Certain restrictions apply to the declaration of ports in a VITAL-compliant entity interface.

VITAL_entity_port_declaration ::=
[**signal**] identifier_list : [mode] type_mark [index_constraint] [:= *static*_expression] ;

The identifiers in an entity port declaration shall not contain underscore characters.

A port that is declared in an entity port declaration shall not be of mode LINKAGE.

The type mark in an entity port declaration shall denote a type or subtype that is declared in package Std_Logic_1164. The type mark in the declaration of a scalar port shall denote the subtype Std_Ulogic or a subtype of Std_Ulogic. The type mark in the declaration of an array port shall denote the type Std_Logic_Vector.

NOTE—The syntactic restrictions on the declaration of a port in a VITAL Level 0 entity are such that the port cannot be a guarded signal. Furthermore, the declaration cannot impose a range constraint on the port, nor can it alter the resolution of the port from that defined in the standard logic package.

4.3.2 Generics

The generics declared in a VITAL Level 0 entity generic clause may be timing generics, control generics, or other generic objects. Timing generics and control generics serve a special purpose in a VITAL-compliant model; specific rules govern their declaration and use. Other generics may be defined to control functionality; such generics are not subject to the restrictions imposed on timing or control generics.

4.3.2.1 Timing generics

This standard defines a number of *timing generics*, which represent specific kinds of timing information. Each kind of timing generic is classified as either a *backannotation timing generic* or a *negative constraint timing generic*, depending on whether the value of the generic is set during the backannotation phase of simulation or the negative constraint calculation phase of simulation. Rules governing the declaration of these generics insure that a mapping can be established between the timing generics of a model and the corresponding SDF timing information or negative constraint delays.

VITAL_timing_generic_declaration ::=
[**constant**] identifier_list ::= [**in**] type_mark [index_constraint] [:= *static*_expression] ;

A timing generic is characterized by its name and its type. The naming conventions (see 4.3.2.1.1) communicate the kind of timing information specified, as well as the port(s) or delay path(s) to which the timing information applies. The type of a timing generic (see 4.3.2.1.2) indicates which of a variety of forms the associated timing value takes.

A VITAL-compliant description may declare any number of timing generics. There are no required timing generics.

Examples:

```
tperiod_Clk  : VITALDelayType   := 5 ns;
```

```
tpd_Clk_Q  : VITALDelayType01  := (tr01 => 2 ns, tr10 => 3 ns);
tipd_D     : VITALDelayType01Z := ( 1 ns, 2 ns, 3 ns, 4 ns, 5 ns, 6 ns )
```

NOTE—The value of a backannotation timing generic is set during the backannotation phase; however, if negative timing constraints are in effect, its value may be adjusted during the subsequent negative constraint calculation phase.

4.3.2.1.1 Timing generic names

The name of a timing generic shall adhere to the naming conventions for timing generics. If the name of a generic does not adhere to these conventions, then the generic is not a timing generic.

The form of a timing generic name and its lexical constituents are described by lexical replacement rules similar to the replacement rules for syntactic constructs. White space is included in these rules to enhance readability; however, white space is not permitted within an identifier. Different elements used to construct names are distinguished by enclosing angle brackets, which are not themselves part of the name. If a lexical element enclosed by angle brackets does not have a replacement rule, then it corresponds to a VHDL identifier described by the text inside the angle brackets. Boldface indicates literal text. Underscores serve as connectors between constituent elements; they are also literal text.

```
<VITALTimingGenericName> ::=
      <VITALBackannotationGenericName>
    | <VITALNegativeConstraintGenericName>

<VITALBackannotationGenericName> ::=
      <VITALPropagationDelayName>
    | <VITALInputSetupTimeName>
    | <VITALInputHoldTimeName>
    | <VITALInputRecoveryTimeName>
    | <VITALInputRemovalTimeName>
    | <VITALInputPeriodName>
    | <VITALPulseWidthName>
    | <VITALInputSkewTimeName>
    | <VITALNoChangeHoldTimeName>
    | <VITALNoChangeSetupTimeName>
    | <VITALInterconnectPathDelayName>
    | <VITALDeviceDelayName>

<VITALNegativeConstraintGenericName> ::=
      <VITALInternalClockDelayName>
    | <VITALInternalSignalDelayName>
    | <VITALBiasedPropagationDelayName>
```

The name of a timing generic is constructed from a timing generic prefix and a number of other elements representing device labels, ports or signals, edges, and conditions. These various elements are combined in a fixed manner, creating three distinct sections of the name: the timing generic prefix, the timing generic port specification, and the timing generic suffix.

A *timing generic prefix* is a lexical element that serves as the beginning of the VHDL simple name of a timing generic. It identifies the kind of timing information that the generic represents, which in turn determines whether the generic is a backannotation timing generic or a negative constraint timing generic. The timing generic prefix consists of the sequence of characters preceding the first underscore in the generic name. It is an error for a model to use a timing generic prefix to begin the simple name of an entity generic that is not a timing generic.

This standard defines the following set of timing generic prefixes:

tpd	tsetup	thold	trecovery	tremoval
tperiod	tpw	tskew	tncsetup	tnchold
tipd	tdevice	ticd	tisd	tbpd

The *timing generic port specification* identifies the port(s) with which the timing data is associated. It may contain both port and instance names. A port that is referenced in a timing generic port specification is said to be *associated* with that timing generic.

The discussion of timing generic names associates timing generics with entity ports; however, a model may use a signal or some other item in place of an entity port. If the port name extracted from a timing generic port specification does not denote a port on the entity, then no assumptions are made about the item denoted by the port name, and no consistency checks are performed between the timing generic and the named item.

Backannotation and negative constraint calculation require the determination of the name(s) of the port(s) associated with a particular timing generic. A port name is *extracted* from the port specification portion of a timing generic name by taking the lexical element corresponding to that port (a sequence of characters that constitute a VHDL identifier, delimited by underscores), as defined by the naming conventions for that sort of a timing generic.

The name of a timing generic may contain a *timing generic suffix* that corresponds to a combination of SDF constructs representing conditions and edges. The forms of these SDF-related suffixes are described by the following rules:

```
<SDFSimpleConditionAndOrEdge> ::=
      <ConditionName>
    | <Edge>
    | <ConditionName>_<Edge>

<SDFFullConditionAndOrEdge> ::=
    <ConditionNameEdge> [ _<SDFSimpleConditionAndOrEdge> ]

<ConditionName> ::=
    simple_name

<Edge> ::=
    posedge
  | negedge
  | 01
  | 10
  | 0z
  | z1
  | 1z
  | z0

<ConditionNameEdge> ::=
      [ <ConditionName>_ ] <Edge>
    | [ <ConditionName>_ ] noedge
```

A condition name is a lexical element that identifies a condition associated with the timing information. The condition name may be mapped to a corresponding condition expression in an SDF file according to the mapping rules described in 5.2.7.3.2.

An edge identifies an edge associated with the timing information. The edge may be mapped to an edge name specified in an SDF file using the mapping rules described in 5.2.7.3.1.

NOTE—It is assumed that the names in timing generic port specifications will generally denote entity ports; however, a model may instead name other items that may or may not be visible from the enclosing entity declaration (internal signals, for instance). If a port name in a timing generic port specification does not denote a port on the entity, then there are no requirements for consistency between the timing generic and the named item (in fact, the named item does not even have to exist); hence, no consistency checks are performed. A tool that processes VITAL-compliant models may choose to issue a warning in this case.

4.3.2.1.2 Timing generic subtypes

The type mark in the declaration of a timing generic shall denote a VITAL delay type or subtype. These are discussed in 9.1.

If each port name in the port specification of a timing generic name denotes an entity port, then the type and constraint of the timing generic shall be consistent with those of the associated port(s). This consistency is defined as follows:

— If the timing generic is associated with a single port and that port is scalar, then the type of the timing generic shall be a scalar form of delay type. If the timing generic is associated with two scalar ports, then the type of the timing generic shall be a scalar form of delay type.

— If a timing generic is declared to be of a vector form of delay type, then it represents delays associated with one or more vector ports. If such a timing generic is associated with a single port and that port is vector, then the type of the timing generic shall be a vector form of delay type, and the constraint on the generic shall match that on the associated port. If the timing generic is associated with two ports, one or more of which is vector, then the type of the timing generic shall be a vector form of delay type, and the length of the index range of the generic shall be equal to the product of the number of scalar subelements in the first port and the number of scalar subelements in the second port.

NOTE—These consistency requirements between timing generic and port(s) do not apply if the port specification in the timing generic identifies an item that is not an entity port. In this case the model assumes responsibility for the appropriate type and constraint for the timing generic.

4.3.2.1.3 Timing generic specifications

Each form of timing generic represents a particular kind of timing information. Additional restrictions on the name and type or subtype may be imposed on generics representing a particular kind of timing information. A description of the acceptable forms for a particular kind of timing generic is provided in the subclause describing that kind of timing generic.

In the following discussion, an *input port* is a VHDL port of mode IN or INOUT. An *output port* is a VHDL port of mode OUT, INOUT, or BUFFER.

4.3.2.1.3.1 Propagation delay

A timing generic beginning with the prefix tpd is a backannotation timing generic representing propagation delay associated with the specified input-to-output delay path. Its name is of the form

<VITALPropagationDelayName> ::=
tpd_<InputPort>_<OutputPort> [_<SDFSimpleConditionAndOrEdge>]

The type of a propagation delay generic shall be a VITAL delay type (see 9.1).

4.3.2.1.3.2 Input setup time

A timing generic beginning with the prefix tsetup is a backannotation timing generic representing the setup time between an input reference port and an input test port. Its name is of the form

<VITALInputSetupTimeName> ::=
tsetup_<TestPort>_<ReferencePort> [_<SDFFullConditionAndOrEdge>]

The type of an input setup time generic shall be a simple VITAL delay type (see 9.1).

4.3.2.1.3.3 Input hold time

A timing generic beginning with the prefix thold is a backannotation timing generic representing the hold time between an input reference signal and an input test signal. Its name is of the form

<VITALInputHoldTimeName> ::=
thold_<TestPort>_<ReferencePort> [_<SDFFullConditionAndOrEdge>]

The type of an input hold time generic shall be a simple VITAL delay type (see 9.1).

4.3.2.1.3.4 Input recovery time

A timing generic beginning with the prefix trecovery is a backannotation timing generic representing the input recovery time between an input test signal and an input reference signal (similar to a setup constraint). Its name is of the form

<VITALInputRecoveryTimeName> ::=
trecovery_<TestPort>_<ReferencePort> [_<SDFFullConditionAndOrEdge>]

The type of an input recovery time generic shall be a simple VITAL delay type (see 9.1).

4.3.2.1.3.5 Input removal time

A timing generic beginning with the prefix tremoval is a backannotation timing generic representing input removal time between an input test signal and an input reference signal (similar to a hold constraint). Its name is of the form

<VITALInputRemovalTimeName> ::=
tremoval_<TestPort>_<ReferencePort> [_<SDFFullConditionAndOrEdge>]

The type of an input removal time generic shall be a simple VITAL delay type (see 9.1).

4.3.2.1.3.6 Period

A timing generic beginning with the prefix tperiod is a backannotation timing generic representing the minimum period time. Its name is of the form

<VITALInputPeriodName> ::=
tperiod_<InputPort> [_<SDFSimpleConditionAndOrEdge>]

If present, the edge specifier indicates the edge from which the period is measured.

The type of a period generic shall be a simple VITAL delay type (see 9.1).

4.3.2.1.3.7 Pulse width

A timing generic beginning with the prefix tpw is a backannotation timing generic representing the minimum pulse width. Its name is of the form

<VITALPulseWidthName> ::=
tpw_<InputPort> [_<SDFSimpleConditionAndOrEdge>]

The edge specifier, if present, indicates the edge from which the pulse width is measured. A posedge specification indicates a high pulse, and a negedge specification indicates a low pulse.

The type of a pulse width generic shall be a simple VITAL delay type (see 9.1).

4.3.2.1.3.8 Input skew time

A timing generic beginning with the prefix tskew is a backannotation timing generic representing skew time between a pair of signals. Its name is of the form

<VITALInputSkewTimeName> ::=
tskew_<FirstPort>_<SecondPort> [_<SDFFullConditionAndOrEdge>]

The type of an input skew generic shall be a simple VITAL delay type (see 9.1).

4.3.2.1.3.9 No-change setup time

A timing generic beginning with the prefix tncsetup is a backannotation timing generic representing the setup time associated with a no-change timing constraint. Its name is of the form

<VITALNoChangeSetupTimeName> ::=
tncsetup_<TestPort>_<ReferencePort> [_<SDFFullConditionAndOrEdge>]

A no-change setup time generic shall appear in conjunction with a corresponding no-change hold time generic.

The type of a no-change setup time generic shall be a simple VITAL delay type (see 9.1).

4.3.2.1.3.10 No-change hold time

A timing generic beginning with the prefix tnchold is a backannotation timing generic representing a hold time associated with a no-change time constraint. Its name is of the form

<VITALNoChangeHoldTimeName> ::=
tnchold_<TestPort>_<ReferencePort> [_<SDFFullConditionAndOrEdge>]

A no-change hold time generic shall appear in conjunction with a corresponding no-change setup time generic.

The type of a no-change hold time generic shall be a simple VITAL delay type (see 9.1).

4.3.2.1.3.11 Interconnect path delay

A timing generic beginning with the prefix tipd is a backannotation timing generic representing the external wire delay to a port, or an interconnect delay that is abstracted as a simple wire delay on the port. Its name is of the form

<VITALInterconnectPathDelayName> ::=
tipd_<InputPort>

The type of an interconnect path delay generic shall be a VITAL delay type (see 9.1).

4.3.2.1.3.12 Device delay

A timing generic beginning with the prefix tdevice is a backannotation generic representing the delay associated with a device (primitive instance) within the cell model. Its name is of the form

<VITALDeviceDelayName> ::=
tdevice_<InstanceName> [_<OutputPort>]

The type of a device delay generic shall be a VITAL delay type (see 9.1).

4.3.2.1.3.13 Internal signal delay

A timing generic beginning with the prefix tisd represents the internal delay for a data or control port and is used to model negative timing constraints (see 8.2). Its name is of the form

<VITALInternalSignalDelayName> ::=
tisd_<InputPort>_<ClockPort>

The type of an internal signal delay generic shall be a scalar form of a simple VITAL delay type (see 9.1).

4.3.2.1.3.14 Biased propagation delay

A timing generic beginning with the prefix tbpd represents a propagation delay that is adjusted to accommodate negative timing constraints (see 8.2). Its name is of the form

<VITALBiasedPropagationDelayName> :=
tbpd_<InputPort>_<OutputPort>_<ClockPort> [_<SDFSimpleConditionAndOrEdge>]

The type of a biased propagation delay generic shall be a VITAL delay type (see 9.1).

There shall exist, in the same entity generic clause, a corresponding propagation delay generic denoting the same ports, condition name, and edge. Furthermore, the type of the biased propagation generic shall be the same as the type of the corresponding propagation delay generic.

4.3.2.1.3.15 Internal clock delay

A timing generic beginning with the prefix ticd represents the internal delay for a clock and is used to model negative timing constraints (see 8.2). Its name is of the form

<VITALInternalClockDelayGenericName> ::=
ticd_<ClockPort>

The type of an internal clock delay generic shall be a scalar form of a simple VITAL delay type (see 9.1).

The name given for the clock port in an internal clock delay generic name is considered to be a *clock signal name.* It is an error for a clock signal name to appear as one of the following elements in the name of a timing generic:

— As either the input port or output port in the name of a biased propagation delay generic

— As the input signal name in an internal signal delay timing generic

— As the test port in a timing check or recovery removal timing generic

4.3.2.2 Control generics

This standard defines a number of *control generics* that provide special support for certain operations.

```
VITAL_control_generic_declaration ::=
    [ constant ] identifier_list ::= [ in ] type_mark [ index_constraint ] [ := static_expression ] ;
```

A control generic is characterized by a name, a type, and an assumed meaning. Definition and use of these generics is not required; however, if a generic in an entity has a control generic name, then that generic is a control generic, and its declaration shall conform to the rules in effect for that kind of control generic. It is erroneous for a model to use a control generic for other than its stated purpose.

A generic with the name InstancePath shall be of the predefined type String. It is the string representation of the full path name of the current instance.

A generic with the name TimingChecksOn shall be of the predefined type Boolean. It may be used to enable timing checks. The value True indicates that timing checks should be enabled.

The XOn and MsgOn generics are switches that may be used as standard mechanisms for control of 'X' generation and assertion message generation relating to timing and glitch violations.

A generic with the name XOn shall be of the predefined type Boolean. It may be used to control 'X' generation for timing checks and path delays. The value True indicates that timing or other violations should cause certain output ports to be assigned the value 'X'.

A generic with the name MsgOn shall be of the predefined type Boolean. It may be used to control the generation of assertion messages for timing constraint violations. The value True indicates that assertion messages should be issued when violations are encountered; the value False means that assertion messages should not be issued.

NOTES

1—The declaration of a control generic by itself has no effect; the generic has to be associated with an appropriate formal parameter of a VITAL standard package subprogram or named in a timing check condition to have the intended effect. Use of a control generic is not limited to these contexts.

2—The XOn and MsgOn generics are similar, but not identical, to the EIA 5670000-91 [B3] XGeneration and MGeneration features. In particular, declaration of an XOn or MsgOn generic does *not* automatically enable timing checks.

4.4 The VITAL Level 0 architecture body

A VITAL Level 0 architecture body defines the body of a VITAL Level 0 design entity.

```
VITAL_Level_0_architecture_body ::=
    architecture identifier of entity_name is
        VITAL_Level_0_ architecture_declarative_part
    begin
        architecture_statement_part
    end [ architecture_simple_name ] ;

VITAL_Level_0_architecture_declarative_part ::=
    VITAL_Level0_attribute_specification { block_declarative_item }
```

The entity associated with a VITAL Level 0 architecture shall be a VITAL Level 0 entity.

4.4.1 Timing generic usage

It is an error if the value of a timing generic is read inside a VITAL Level 0 model prior to the initialization phase of simulation.

NOTE—There is no requirement that the usage of a timing generic be consistent with the kind of timing information implied by the generic name.

5. Backannotation

The sole point of entry of timing information into a VITAL-compliant model is through the timing generics. With the exception of the use of VITAL_Timing routines, all timing calculations are performed outside of the VHDL model, and external timing information is passed to the model through the backannotation timing generics. *Backannotation* is the process of updating the backannotation timing generics with the external timing information. Signal delays that are used to model negative timing constraints are computed in the negative constraint calculation stage of simulation; their calculation is not part of the backannotation process.

The rules governing the backannotation of timing values into a VITAL-compliant model and the mapping of SDF constructs to backannotation timing generics define the semantics assumed by models that adhere to the VITAL Level 0 specification.

5.1 Backannotation methods

There are two methods for annotating model instances with timing data: through the use of an appropriate VHDL configuration declaration and through the direct import of timing data from one or more SDF files. An appropriate VHDL configuration declaration can be generated from SDF data or by some other means. If a VITAL-compliant model derives its timing information from SDF data, then the state of that model at the beginning of simulation shall be the same, regardless of the annotation path employed.

NOTE—It is assumed that an SDF file will be created (possibly by a tool such as a delay calculator) using information that is consistent with the library data (e.g., a VHDL model). This implies that, in general, the data in the SDF file will be consistent with that in a corresponding VITAL-compliant model.

5.1.1 Direct SDF import

Direct SDF import is accomplished by reading delay data from one or more SDF files and using this information to modify the backannotation timing generics in a VITAL-compliant model directly. The modification of the backannotation timing generics occurs in the *backannotation phase* of simulation, which directly follows elaboration and directly precedes negative constraint delay calculation. Once the values of the backannotation timing generics have been established and set by the backannotation process, no further modification is permitted except during the negative constraint calculation stage.

The SDF mapping rules are such that an SDF annotator that performs direct SDF import is responsible for insuring the semantic correctness of the association of delay values with backannotation timing generics. As a consequence, a delay value or a set of delay values shall be appropriate for the type class of the corresponding VHDL timing generic, and all applicable VHDL constraints on the value or set of values shall be satisfied.

5.1.2 The SDF annotator

The term *SDF annotator* refers to any tool in the class of tools capable of performing backannotation from SDF data in a VITAL-compliant manner. This class includes tools that generate appropriate configuration declarations from SDF data.

An SDF annotator shall annotate the backannotation timing generics. Furthermore, it shall report an error upon

encountering any form of noncompliance with a requirement in this standard related to the SDF mapping or backannotation process. Its behavior after reporting an error is implementation defined.

Certain SDF constructs are not supported by this standard; these constructs are said to be *unsupported.* Unsupported constructs do not result in the modification of backannotation timing generics, nor do they have any other effect on the backannotation process.

If the SDF data fails to provide a value for a backannotation timing generic in a VITAL-compliant model, then the value of that timing generic shall not be modified during the backannotation process, and the value that was set during standard VHDL elaboration shall remain in effect.

NOTE—A VITAL SDF annotator can also annotate generics other than backannotation timing generics (for example, the InstancePath generic). A VITAL SDF annotator is neither required to annotate nor prohibited from annotating generics on models that are not VITAL compliant.

5.2 The VITAL SDF map

The VITAL SDF map specifies the mapping between specific SDF constructs and the corresponding VHDL timing generics and their values. Some SDF constructs are mapped directly to specific kinds of timing generics or their timing values, others map to lexical elements that can be used to construct any timing generic name, and others identify items in the VHDL design hierarchy (such as instances or ports) to which timing data is applied.

The name of the corresponding VHDL timing generic is determined according to the rules of the VITAL SDF map. It is an error if there is no translation of a supported SDF construct to a legal VHDL identifier. It is an error if the generic name that the SDF annotator constructs from the SDF file is not present in the VHDL model.

The following discussion uses portions of the BNF from the Standard Delay Format Specification to describe SDF constructs. An italicized lowercase identifier represents a SDF syntax construct. The definition of a syntax construct is indicated by the symbol ::=, and alternative definitions are separated by the symbol ||=. Keywords appear in boldface capital letters. Uppercase identifiers represent variable symbols. The form "item?" represents an optional item. The form "item*" represents zero or more occurrences of the item. The form "item+" indicates one or more occurrences of the item.

5.2.1 Delay file

An SDF delay file consists of a header and a sequence of one or more cells containing timing data.

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

The information in each SDF cell of each SDF file is mapped to the corresponding VHDL constructs using general information, such as the time scale, found in the corresponding SDF header.

5.2.2 Header section

The SDF annotator uses the information in the SDF header to read and interpret the SDF file correctly. In general, the entries in the header section have no direct effect on the backannotation process itself. Some header entries are purely informational, and others (those detailed below) provide information needed by the SDF annotator to interpret the SDF file correctly.

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

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

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

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

The *sdf_version* shall refer to the Standard Delay Format Specification, Version 2.1.

The *hierarchy_divider* identifies which lexical character (a period or a slash) separates elements of a hierarchical SDF path name.

The *time_scale* determines the time units associated with delay values in the file.

5.2.3 CELL entry

The SDF CELL entry associates a set of timing data with one or more instances in a design hierarchy.

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

The timing data in the CELL entry is mapped to a VHDL model as follows:

a) The *cell_instance* and *celltype* constructs are used to locate a path or a set of paths in the VHDL design hierarchy that correspond to the instance(s) to which the data applies.

b) Each supported timing specification in the sequence of *timing_spec* constructs is mapped to the backannotation timing generic(s) of the specified instance(s) in the VHDL design hierarchy, and the corresponding timing data is transformed into value(s) appropriate for the generic(s).

The CORRELATION entry is not supported by this standard.

5.2.4 INSTANCE and CELLTYPE entries

The SDF cell instance, in conjunction with the SDF cell type, identifies a set of VHDL component instances to which the timing data in a CELL entry applies. This set is constructed by identifying the VHDL component instances (specified by the SDF cell instance) that match the component type (specified by the SDF cell type).

The CELLTYPE entry

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

indicates that the timing data is applicable only to those component instances that correspond to a VHDL component having the name that is specified by the QSTRING variable. Such instances are said to *match* the cell type.

An SDF cell instance is a sequence of one or more INSTANCE entries that name a path or set of paths in the design hierarchy.

cell_instance ::= *instance*+
| (**INSTANCE** WILDCARD)

instance ::= (**INSTANCE** PATH?)

The first form of cell instance names the path of a particular instance or set of instances. The second form of SDF cell instance is a wildcard that identifies all component instances, in or below the current level of the design hierarchy, that match the cell type.

A VHDL instance described by one or more SDF instance paths is located by mapping each successive path element of the PATH variable of each successive INSTANCE entry to a VHDL block, generate, or component instantiation statement label of the same name at the next level of the design hierarchy, beginning at the level at which the SDF file is applied. Path elements within an SDF PATH IDENTIFIER are separated by hierarchy divider characters. It is an error if, at any level, an appropriate VHDL concurrent statement label does not exist for the corresponding SDF path element. The last path element shall denote a component instance (i.e., it cannot denote a block or generate statement). The VHDL component associated with the instance shall match the cell type.

An SDF path element may contain a bit spec of the form [*integer*] or [*integer*:*integer*]. Such a path element corresponds to one or more expansions of a FOR generate statement. A bit spec containing a single integer corresponds to a single expansion of the generate statement, and a bit spec containing a pair of integers corresponds to a set of expansions described by a range. It is an error if the alphanumeric portion of a path element containing a bit spec does not correspond to the label of a FOR generate statement.

The set of generate statement expansions corresponding to an SDF path element containing a bit spec shall be determined by mapping the SDF integer or pair of integers to the appropriate VHDL index or range. The VHDL value corresponding to a bit spec integer is obtained by mapping the bit spec integer to the VHDL value whose position number is that bit spec integer — base_type'VAL(*integer*), where base_type is the base type of the generate parameter. It is an error if the corresponding VHDL value does not belong to the discrete range specified in the generate statement. The VHDL range corresponding to a pair of integers is constructed by mapping the left and right SDF integers to the corresponding VHDL values representing the left and right bounds, respectively, and then selecting a direction that results in a range that is not a null range.

Example:

The SDF entry

```
(CELL
    (CELLTYPE "DFF")
    (INSTANCE a1.b1.c1)
    (DELAY
        (ABSOLUTE (IOPATH i1 o1 (10) (20)))
    )
)
```

requires the SDF annotator to look for the concurrent statement with label a1 at the current level and b1 and c1 at the successive levels below a1. Level c1 shall be the label of a component instantiation statement, and backannotation is performed on the timing generics of c1.

5.2.5 Timing specifications

An SDF timing specification contains delay or timing constraint data that is mapped directly to one or more backannotation timing generics.

timing_spec ::= *del_spec*
||= *tc_spec*

An SDF timing specification consists of a data value (or a set of data values) and a number of constructs describing the nature of the data value(s). The constituents of the timing specification are mapped to different, but related, VHDL items. The data value or set of data values is mapped to an appropriate VHDL delay value. The remainder of the timing specification is mapped to a specific timing generic name or pair of names.

5.2.6 Data value mapping

The timing information in an SDF timing specification is specified in terms of *value*, *rvalue*, and *rvalue_list* constructs. A *value* or *rvalue* can consist of one, two, or three data values corresponding to the minimum, typical, and maximum value. However, for annotation to VITAL designs, only one of these values is used. An SDF annotator shall provide a mechanism for selecting one value from the triple of values.

The type of the timing generic determines the type of the VHDL delay value to which the corresponding SDF timing information is mapped. It is an error if a backannotation timing generic holds fewer delay values than the number specified in the corresponding SDF entry. If a backannotation timing generic is of a transition-dependent delay type that contains more values than are specified by the corresponding SDF entry, then the SDF annotator supplies the remaining delays in the transition-dependent delay value according to a predefined mapping.

A simple SDF *value* or *rvalue* is mapped to an equivalent VHDL value of type Time.

An *rvalue_list* can contain one, two, three, six, or twelve *rvalues* after SDF extension (in which lists of an intermediate length are interpreted as though they had trailing empty parentheses). If the timing generic is of a scalar form of simple delay type, then the corresponding *rvalue_list* shall contain a single *rvalue*, and the resulting VHDL delay value is a single value of type Time. Otherwise, the timing generic shall be of a scalar form of transition-dependent delay type, and the VHDL delay value is constructed by filling each element of the array with the appropriate SDF value, according to the mapping described in 5.2.7. In table 1, each row represents a form of SDF *rvalue_list*, and each column represents the corresponding delay value for a particular transition.

Table 1—Mapping of SDF delay values to VITAL transition-dependent delay values

SDF *rvalue_list*	VITAL transition-dependent delay value											
	tr01	**tr10**	**tr0z**	**trz1**	**tr1z**	**trz0**	**tr0x**	**trx1**	**tr1x**	**trx0**	**trxz**	**trzx**
v1	v1	v1	v1	v1	v1	v1	–	–	–	–	–	–
v1 v2	v1	v2	v1	v1	v2	v2	–	–	–	–	–	–
v1 v2 v3	v1	v2	v3	v1	v3	v2	–	–	–	–	–	–
v1 v2 v3 v4 v5 v6	v1	v2	v3	v4	v5	v6	–	–	–	–	–	–
v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10	v11	v12

NOTE—An SDF annotor follows the SDF annotation rules regarding null delay values and extension of lists of rvalues.

5.2.7 Mapping to timing generics

The form of the VHDL timing generic name corresponding to an SDF timing specification is determined by the nature of the timing information and by other items, such as ports, that the timing specification references.

The fact that SDF is case sensitive and VHDL is not case sensitive may cause SDF names that differ only in case to map to the same VHDL generic name. Such cases are treated as multiple SDF entries for the same generic.

5.2.7.1 DELAY entry

Different kinds of SDF DELAY entries are mapped to different kinds of backannotation timing generics.

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

The delay types PATHPULSE and GLOBALPATHPULSE are not supported by this standard, nor is the NETDELAY delay definition.

5.2.7.1.1 ABSOLUTE and INCREMENT delay

SDF delay data is designated as incremental or absolute through the form of the delay type construct, *deltype*. The delay type determines how the timing data is applied to the corresponding timing generic(s).

deltype ::= (**ABSOLUTE** *del_def+*)
||= (**INCREMENT** *del_def+*)

During backannotation, delay values are applied sequentially, in the order that they appear in the SDF file. An absolute delay replaces an existing generic value. An incremental delay value is added to the existing value of the generic.

NOTE—If more than one SDF delay or timing constraint entry maps to the same generic name, the SDF annotator updates their values using their existing values when appropriate. For example, if the first entry results in updating the value of a certain generic, and a subsequent SDF entry with INCREMENT delays maps to the same generic name, then the new value of the generic is determined by incrementing the previously updated generic value.

5.2.7.1.2 IOPATH delay

The SDF path delay entry can take a simple form or a conditional form.

del_def ::= (**IOPATH** *port_spec port_path rvalue_list*)
||= (**COND** *conditional_port_expr* (**IOPATH** *port_spec port_path rvalue_list*))

Each maps to a propagation delay generic (see 4.3.2.1.3.1) of the form

tpd_<InputPort>_<OutputPort> [_<SDFSimpleConditionAndOrEdge>]

The <InputPort> corresponds to the port name specified in the *port_spec,* and the <OutputPort> corresponds to the port specifed in the *port_path.* The <SDFSimpleConditionAndOrEdge> is derived from the *conditional_port_expr,* if present, and the *port_edge* of the *port_spec*, if present.

Example:

The SDF entry

```
(IOPATH Input Output (10) (20))
```

maps to the generic

```
tpd_Input_Output
```

5.2.7.1.3 PORT delay

The SDF port entry

del_def ::= (**PORT** *port_path rvalue_list*)

maps to an interconnect path delay generic (see 4.3.2.1.3.11) of the form

tipd_<InputPort>

The <InputPort> corresponds to the port specified in the *port_path*.

Example:

The SDF entry

```
(PORT Input (10) (20))
```

maps to the generic

```
tipd_Input
```

5.2.7.1.4 INTERCONNECT delay

The SDF interconnect delay entry

del_def ::= (**INTERCONNECT** *port_instance port_instance rvalue_list*)

port_instance ::= *port_path*
::= *instance port_path*

maps to an interconnect path delay generic (see 4.3.2.1.3.11) of the form

tipd_<InputPort>

The <InputPort> corresponds to the port specified in the second *port_instance*. The *instance* constructs in this case do not contribute to the corresponding VHDL timing generic name.

If more than one SDF entry maps to the same interconnect path delay generic, then it is assumed that more than one port drives the specified input port. For such cases, the SDF annotator shall provide an option to select between the last, the maximum, and the minimum delay values.

Example:

The SDF entry

```
(INTERCONNECT Output Input (10) (20))
```

maps to the generic

```
tipd_Input
```

5.2.7.1.5 DEVICE delay

The SDF device entry can represent the delay on the output of a modeling primitive (used inside an ASIC cell) or the delay on the output of an entire ASIC cell. This standard supports only those device entries that specify the delays across primitives used inside a VITAL model. See clause 7 for a description of the available primitives.

del_def ::= (**DEVICE** *port_instance*? *rvalue_list*)

port_instance ::= *port_path*
::= *instance port_path*

The device entry maps to a device delay generic (see 4.3.2.1.3.12) of the form

tdevice_<InstanceName> [_<OutputPort>]

The <InstanceName> is derived from the name of the SDF instance to which the DEVICE entry applies (it is not derived from the *port_instance* construct). If the SDF instance has a hierarchical name, the lowest level instance name is the <InstanceName>. The optional <OutputPort> is present if *port_instance* is specified.

NOTE—It is expected that the specified <InstanceName> will be the label for a VITAL primitive concurrent procedure call.

Example:

The SDF entry

```
(CELL  (CELLTYPE "AN2")
       (INSTANCE Top.I1.P1)
       (DELAY (ABSOLUTE (DEVICE Z  (10) (20))))
)
```

maps to the generic

tdevice_P1_Z

5.2.7.2 TIMINGCHECK entry

The SDF timing check entry defines a number of timing checks and timing constraints.

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

tc_def ::= *tchk_def*
||= *cns_def*

tchk_def ::= (**SETUP** *port_tchk port_tchk rvalue*)
||= (**HOLD** *port_tchk port_tchk rvalue*)
||= (**SETUPHOLD** *port_tchk port_tchk rvalue rvalue*)
||= (**RECOVERY** *port_tchk port_tchk rvalue*)
||= (**SKEW** *port_tchk port_tchk rvalue*)
||= (**WIDTH** *port_tchk value*)
||= (**PERIOD** *port_tchk value*)
||= (**NOCHANGE** *port_tchk port_tchk rvalue rvalue*)

SDF timing constraint entries for forward annotation—*cns_def* entries PATHCONSTRAINT, SUM, DIFF, and SKEWCONSTRAINT—are not supported by this standard.

Each true timing check definition (i.e., each *tc_def* that is a *tchk_def*) is mapped to one or more VHDL backannotation timing generics. In general, there is a one-to-one correspondence between SDF timing check definitions and VHDL backannotation timing generics. The SDF SETUPHOLD and NOCHANGE constructs are exceptions. Each is processed as though it were replaced by the collectively equivalent setup and hold entries. Hence, the SDF timing check is mapped to two separate VHDL backannotation timing generics—one each for setup and hold times.

An SDF timing check definition can contain one or two timing check port specifications (*port_tchk*s), which may

be further modified with condition and edge constructs. The corresponding VHDL generic name for an SDF timing check is constructed from the timing check type and the timing check port specifications in the manner shown in the following list.

a) The appropriate generic timing prefix is selected using the following mapping:

SETUP	tsetup	(see 4.3.2.1.3.2)
HOLD	thold	(see 4.3.2.1.3.3)
SETUPHOLD	tsetup	
	thold	
RECOVERY	trecovery	(see 4.3.2.1.3.4)
SKEW	tskew	(see 4.3.2.1.3.8)
WIDTH	tpw	(see 4.3.2.1.3.7)
PERIOD	tperiod	(see 4.3.2.1.3.6)
NOCHANGE	tncsetup	(see 4.3.2.1.3.9)
	tnchold	(see 4.3.2.1.3.10)

b) The appropriate timing generic port specification is added to the generic name as follows: In the order in which they appear in the SDF entry, the *port* in each *port_tchk* is mapped to the corresponding VHDL timing generic port specification and the result appended to the timing generic name with a preceding underscore.

c) If a *port_tchk* in the timing check definition contains an edge or a condition, then the appropriate timing generic suffix is constructed according to the rules in 5.2.7.2.1 and appended to the timing generic name with a preceding underscore.

5.2.7.2.1 Condition and edge combinations

A *port_tchk* construct in an SDF timing check definition can contain a condition (the *timing_check_condition*) or an edge (in the form of the EDGE_IDENTIFIER in a *port_spec* that is a *port_edge*). A timing check definition can contain one or two *port_tchk* specifications. The conditions and edges associated with these ports are mapped to a timing generic suffix that is appended to the timing generic name with a preceding underscore.

port_tchk ::= *port_spec*
||= (**COND** *timing_check_condition port_spec*)

The conditions and edges in a timing check definition that is associated with a single port (i.e., a PERIOD or WIDTH entry) map to the <SDFSimpleConditionAndOrEdge> lexical suffix that is derived from the *timing_check_condition*, if present, and the EDGE_IDENTIFIER of the *port_spec*, if present.

Each of the remaining timing check definitions is associated with a pair of ports, and the conditions and edges map to the <SDFFullConditionAndOrEdge> lexical suffix

<ConditionNameEdge> [_<SDFSimpleConditionAndOrEdge>]

The <ConditionNameEdge> portion is derived from the first *port_tchk* construct. If the first *port_tchk* does not have an edge, then the <ConditionNameEdge> is of the form

[<ConditionName>_] **noedge**

Otherwise, the <ConditionNameEdge> is derived from the *timing_check_condition*, if present, and the EDGE_IDENTIFIER of the *port_spec*, if present.

The <SDFSimpleConditionAndOrEdge> is derived from the second *port_tchk* construct, using the

timing_check_condition, if present, and the EDGE_IDENTIFIER of the *port_spec*, if present.

Examples:

The SDF entry

```
(COND RESET == 1'b1 && CLK == 1'b1 (IOPATH posedge A Y (10) (20) ) )
```

maps to the VHDL identifier

```
tpd_A_Y_RESET_EQ_1_AN_CLK_EQ_1_posedge
```

The SDF entry

```
(SETUP (COND Reset == 1'b1 DATA) (posedge CLK) (5))
```

maps to the VHDL identifier

```
tsetup_DATA_CLK_RESET_EQ_1_noedge_posedge
```

5.2.7.3 Mapping of SDF constructs to general VHDL lexical elements

Certain SDF constructs are not themselves mapped to a specific kind of timing generic; instead, they are mapped to lexical elements that may be used to construct any backannotation timing generic name. These general SDF constructs include edges, conditions, and port specifications.

5.2.7.3.1 Edges

The SDF edge construct maps to a VHDL lexical suffix that is textually equivalent to the EDGE_IDENTIFIER.

```
EDGE_IDENTIFIER ::= posedge
                |= negedge
                |= 01
                |= 10
                |= 0z
                |= z1
                |= 1z
                |= z0
```

This lexical suffix is attached to the VHDL generic identifier with an underscore. The location of the lexical suffix within the generic identifier is determined by the context of the edge.

5.2.7.3.2 Conditions

The SDF conditional construct—identified by the keyword COND—can appear in two different contexts: in a conditional path delay of the form

(**COND** *conditional_port_expr* (**IOPATH** *port_spec port_path rvalue_list*))

and as a *port_tchk* specification in a timing check definition

(**COND** *timing_check_condition port_spec*)

In either case, the condition is mapped to a a legal VHDL lexical representation of the conditional expression that is then used to construct a suffix for a timing generic name.

The VHDL lexical suffix is constructed from the conditional expression (the *conditional_port_expr* or *timing_check_condition*) using the following algorithm:

a) Separate each part of the condition with an underscore, removing any white space.

b) Replace each SCALAR_CONSTANT symbol as follows:

1'b0, 1'B0, 'b0, 'B0, 0 by 0

1'b1, 1'B1, 'b1, 'B1, 1 by 1

c) Replace each operator symbol as follows:

(	by OP
)	by CP
{	by OB
}	by CB
[	by OSB
]	by CSB
,	by CM
?	by QM
:	by CLN (only in expr ? expr : expr statements)
+	by PL
-	by MI
*	by MU
/	by DI
%	by MOD
==	by EQ
!=	by NE
===	by EQ3
!==	by NE3
&&	by AN
||	by OR
<	by LT
<=	by LE
>	by GT
>=	by GE
&	by ANB
|	by ORB
^	by XOB
^~	by XNB
~^	by XNB
>>	by RS
<<	by LS
!	by NT
~	by NTB
~&	by NA
~|	by NO

d) Replace each range as follows:

[x:y] by xTOy
[x] by x
where x and y represent indices

Example:

The SDF entry

```
(COND RESET == 1'b1 && CLK == 1'b1 (IOPATH A Y (10) (20) ) )
```

is mapped to the VHDL generic

```
tpd_A_Y_RESET_EQ_1_AN_CLK_EQ_1
```

5.2.7.3.3 Ports

The SDF *port_spec* construct names a port, and possibly an edge, associated with a timing value. An edge, if present, is processed separately and is not necessarily adjacent to the corresponding port name in the resulting VHDL identifier. Processing of edges is discussed in the appropriate contexts.

port_spec ::= *port_path*
||= *port_edge*

port_path ::= *port*
||= PATH HCHAR *port*

port_edge ::= (EDGE_IDENTIFIER *port_path*)

port ::= *scalar_port*
||= *bus_port*

scalar_port ::= IDENTIFIER
||= IDENTIFIER [DNUMBER]

bus_port ::= IDENTIFIER [DNUMBER : DNUMBER]

The form of the SDF port name may impose certain requirements on the subtype of the corresponding VHDL generic. For backannotation purposes, the hierarchical form of a *port_path* is equivalent to its simple form, *port*.

The IDENTIFIER in a *port* maps to a VHDL name that is textually equivalent to the IDENTIFIER. The result shall be a legal VHDL identifier.

If the *port* is of one of the forms

IDENTIFIER [DNUMBER]
IDENTIFIER [DNUMBER : DNUMBER]

then it is assumed that the corresponding VHDL port is a vector, and the associated timing generic shall be of a vector form of VITAL delay type. It is an error if a backannotation timing generic is of a vector form of delay type and none of the corresponding ports in the SDF file have an index or range specification.

Each *port* entry with an index or range specification maps to an element or set of elements in the corresponding timing generic array value. Each such element is denoted by an index, which is derived from the SDF index or range specification according to the rules in the following subclauses. It is an error if the array index of the generic element corresponding to a particular SDF entry is out of range for that generic.

NOTE—An escape character in an SDF IDENTIFIER is an error because it cannot be mapped to a legal VHDL identifier.

5.2.7.3.3.1 Mapping of a single bus port

The scheme outlined in this subclause applies to an SDF entry that has a single port specification that has an index or range specification. This scheme also applies to an SDF entry that has two port specifications, only one of which has an index or range specification. For any port having an index specification, let

c	denote the index of the SDF port
j1 , *j2*	denote the left and right indices of the corresponding port in the VHDL model
g1 , *g2*	denote the left and right indices of the corresponding generic in the VHDL model

For the SDF port index *c*, the SDF annotator computes the corresponding index *x* in the generic array by using the following row-dominant scheme:

$$x = g2 + \text{abs}(c - j2) \times (g1 - g2)/\text{abs}(g1 - g2)$$

If the SDF port has a range specification rather than an index specification, then this computation is performed for each delay value in the range of the SDF port.

NOTE—For a generic with a descending range constraint of (g1 **downto** g2), the index computation reduces to

x = g2 + abs(c - j2)

and for a generic with an ascending range constraint of (g1 **to** g2), the index computation reduces to

x = g2 - abs(c - j2)

5.2.7.3.3.2 Mapping of two bus ports

The scheme outlined in this subclause applies to an SDF entry that has two port specifications, both of which have an index or range specification. Let

r, *c*	denote the index of the respective ports in the SDF entry
i1 , *i2*	denote the left and right indices of the VHDL port corresponding to the first SDF port
j1 , *j2*	denote the left and right indices of the VHDL port corresponding to the second SDF port
g1 , *g2*	denote the left and right indices of the corresponding generic in the VHDL model

For the SDF entry having index *r* and *c* as the indices for the first and second ports, the SDF annotator computes the corresponding index *x* in the generic array by using the following row-dominant scheme:

$$x = g2 + (\text{abs}(c - j2) + \text{abs}(r - i2) \times (\text{abs}(j1 - j2) + 1)) \times (g1 - g2)/\text{abs}(g1 - g2)$$

If one or more of the SDF ports has a range specification rather than an index specification, then this computation is performed for each delay value in the range of the SDF ports.

NOTE—For a generic with a descending range constraint of (g1 **downto** g2), the index computation reduces to

x = g2 + abs(c - j2) + abs(r - i2) × (abs(j1 - j2) + 1)

and for a generic with an ascending range constraint of (g1 **to** g2), the index computation reduces to

x = g2 - abs(c - j2) - abs(r - i2) × (abs(j1 - j2) + 1)

Example:

Assuming the following VHDL declarations

```
generic (tpd_A_Y : VitalDelayArrayType01 (0 to 3) := (others => (0 ns, 0 ns)));
port (A : IN std_logic_vector (0 to 1);
      Y : OUT std_logic_vector (1 to 2));
```

the SDF entry

```
(IOPATH   A[0]    Y[1] (10) (20))
```

will cause the SDF annotator to annotate the delay value (10, 20) onto the generic subelement tpd_A_Y(0), and the SDF entry

```
(IOPATH   A[0:1]  Y[1:2] (10) (20))
```

will cause the SDF annotator to annotate the delay value (10, 20) onto the generic subelements tpd_A_Y(0), tpd_A_Y(1), tpd_A_Y(2), and tpd_A_Y(3).

6. The VITAL Level 1 specification

The VITAL Level 1 specification is a set of modeling rules that constrains the descriptions of cell models in order to facilitate the optimization of the set-up and execution of the models, leading to higher levels of performance than could be expected through the acceleration of the basic capabilities provided by the VITAL standard packages alone.

A VITAL Level 1 model description defines an ASIC cell in terms of functionality, wire delay propagation, timing constraints, and output delay selection and scheduling.

6.1 The VITAL_Level1 attribute

A VITAL Level 1 architecture is identified by its decoration with the VITAL_Level1 attribute, which indicates an intention to adhere to the VITAL Level 1 specification.

VITAL_Level1_attribute_specification ::= attribute_specification

A VITAL Level 1 architecture shall contain a specification of the VITAL_Level1 attribute corresponding to the declaration of that attribute in package VITAL_Timing. The entity specification of the decorating attribute specification shall be such that the enclosing architecture inherits the VITAL_Level1 attribute. The expression in the VITAL_Level1 attribute specification shall be the Boolean literal True.

Example:

```
attribute VITAL_Level1 of VitalCompliantArchitecture : architecture is True;
```

6.2 The VITAL Level 1 architecture body

A VITAL Level 1 architecture body defines the body of a VITAL Level 1 design entity.

```
VITAL_Level_1_architecture_body ::=
    architecture identifier of entity_name is
        VITAL_Level_1_architecture_declarative_part
    begin
        VITAL_Level_1_architecture_statement_part
    end [ architecture_simple_name ] ;
```

A VITAL Level 1 architecture shall adhere to the VITAL Level 0 specification, except for the declaration of the VITAL_Level0 attribute.

The entity associated with a VITAL Level 1 architecture shall be a VITAL Level 0 entity. Together, these design units comprise a *VITAL Level 1 design entity*.

The only signals that shall be referenced in a VITAL Level 1 design entity are entity ports and internal signals. References to global signals and signal-valued attributes are not allowed. Each signal declared in a VITAL Level 1 design entity shall have at most one driver.

The use of subprogram calls and operators in a VITAL Level 1 architecture is limited. The only operators or subprograms that shall be invoked are those declared in package Standard, package Std_Logic_1164, or the VITAL standard packages. Formal subelement associations and type conversions are prohibited in the associations of a subprogram call.

6.3 The VITAL Level 1 architecture declarative part

The VITAL Level 1 architecture declarative part contains declarations of items that are available for use within the VITAL Level 1 architecture.

```
VITAL_Level_1_architecture_declarative_part ::=
    VITAL_Level1_attribute_specification
    { VITAL_Level_1_block_declarative_item }

VITAL_Level_1_block_declarative_item ::=
      constant_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | VITAL_internal_signal_declaration
```

6.3.1 VITAL internal signals

A signal that is declared in the declarative part of an architecture is an *internal signal.*

```
VITAL_internal_signal_declaration ::=
    signal identifier_list : type_mark [ index_constraint ] [ := expression ] ;
```

The type mark in the declaration of an internal signal shall denote the standard logic type Std_Ulogic or Std_Logic_Vector.

6.4 The VITAL Level 1 architecture statement part

The statement part of a VITAL Level 1 architecture is a set of one or more concurrent statements that perform specific VITAL activities.

```
VITAL_Level_1_architecture_statement_part ::=
    VITAL_Level_1_concurrent_statement { VITAL_Level_1_concurrent_statement }

VITAL_Level_1_ concurrent_statement ::=
      VITAL_wire_delay_block_statement
    | VITAL_negative_constraint_block_statement
    | VITAL_process_statement
    | VITAL_primitive_concurrent_procedure_call
```

A VITAL Level 1 architecture shall contain at most one wire delay block statement.

If the entity associated with a VITAL Level 1 architecture declares one or more timing generics representing internal clock or internal signal delay, then negative constraints are in effect, and the VITAL Level 1 architecture shall contain exactly one negative constraint block to compute the associated signal delays.

A VITAL Level 1 architecture shall contain at least one VITAL process statement or VITAL primitive concurrent procedure call.

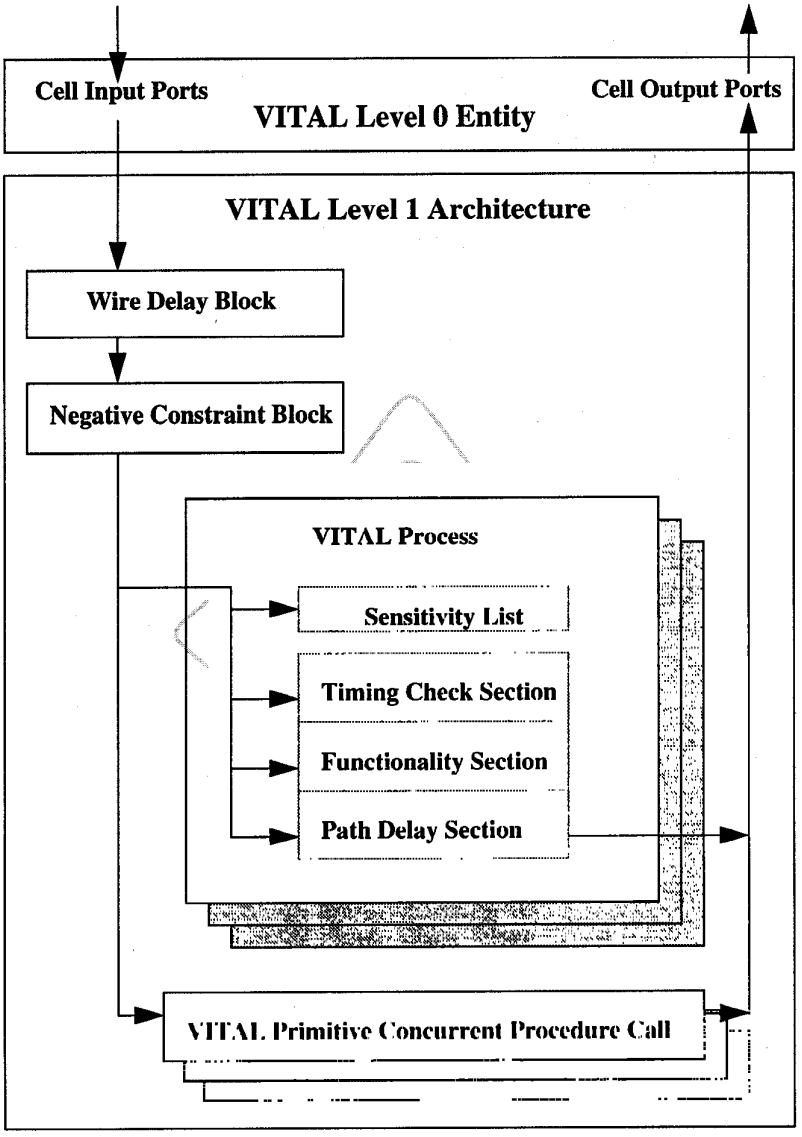


Figure 1—Structure of a VITAL Level 1 model

6.4.1 Wire delay block statement

Interconnect delay between ASIC cells is modeled as an approximation of wire delays at input ports. Wire delays external to a model are propagated inside the model through a *VITAL wire delay block*.

VITAL_wire_delay_block_statement ::=
*block*_label :
block

```
    begin
        VITAL_wire_delay_block_statement_part
    end block [ block_label ] ;

VITAL_wire_delay_block_statement_part ::=
    { VITAL_wire_delay_concurrent_procedure_call
    | VITAL_wire_delay_generate_statement }

VITAL_wire_delay_generate_statement ::=
    generate_label :
    for VITAL_wire_delay_generate_parameter_specification generate
        { VITAL_wire_delay_concurrent_procedure_call }
    end generate [ generate_label ] ;

VITAL_wire_delay_generate_parameter_specification ::=
    identifier in range_attribute_name

VITAL_wire_delay_concurrent_procedure_call ::= concurrent_procedure_call
```

The label of a VITAL wire delay block shall be the name WireDelay.

A wire delay is modeled by a concurrent procedure call, which invokes one of the VitalWireDelay procedures that are declared in the package VITAL_Timing. A VitalWireDelay procedure delays an input signal by a specified delay value using a transport delay. A wire delay block is the only context in which a call to a VitalWireDelay procedure is allowed.

A port that is associated with a wire delay concurrent procedure call is said to have an *associated wire delay*. A wire delay block shall contain at most one wire delay for each port of mode IN or INOUT declared in the VITAL Level 1 design entity.

Associated with each external wire delay is an internal signal representing the delayed port; this internal signal is called the *wire delayed signal*. The subtype of the wire delayed signal shall be the same as that of the corresponding port.

The value of a port with an associated wire delay shall be read only in those contexts that are directly related to the modeling of the wire delay itself; that is, the value of the port shall be read only in the context of the actual parameter part of the wire delay concurrent procedure call. The value of the corresponding wire delayed signal is read elsewhere in the model.

A wire delay is applied at the scalar level. Wire delay for a scalar port is modeled with a simple concurrent procedure call.

Wire delay for an array port is modeled with a generate statement of a specific form. The generate statement shall have a generate parameter specification in which the discrete range is a predefined Range attribute, and the prefix of that attribute shall denote the port with the associated wire delay. The only statement within the generate statement shall be a wire delay concurrent procedure call for an element of the port named in the generate parameter specification. The index selecting the element shall be a name denoting the generate parameter.

The actual parameter part of a wire delay concurrent procedure call shall satisfy the following requirements:

— The actual part associated with the input parameter InSig shall be a name denoting a port of mode IN or INOUT.

— The actual part associated with the output parameter OutSig shall be a name denoting an internal signal that satisfies the requirements for a wire delayed signal.

— The actual part associated with the delay value parameter TWire shall be either a locally static value or a name denoting an interconnect path delay timing generic. The delay value shall be nonnegative.

NOTE—The restrictions on reading the value of a port with an associated wire delay do not preclude the use of the name of the port as a prefix to certain predefined attributes. Use of attributes such as the RANGE attribute may be necessary to declare an appropriate wire delayed signal or to specify an appropriate range in a generate parameter specification.

6.4.2 Negative constraint block statement

A *negative constraint block* is a special form of a VHDL block statement that is required to model negative timing constraint values (see 8.2 for details on modeling negative timing constraints).

```
VITAL_negative_constraint_block_statement ::=
    block_label :
    block
    begin
        { VITAL_negative_constraint_concurrent_procedure_call }
    end block [ block_label ] ;
```

The label of a VITAL negative constraint block shall be the name SignalDelay.

A negative constraint block shall contain exactly one negative constraint concurrent procedure call for each timing generic representing an internal clock delay or an internal signal delay.

A negative constraint concurrent procedure call invokes the procedure VITALSignalDelay that is declared in the package VITAL_Timing. The effect of this call is to delay the associated input port by creating a corresponding internal signal that is delayed by the appropriate amount. A negative constraint block is the only context in which a call to VITALSignalDelay is allowed.

The formal parameters of the VITALSignalDelay procedure are associated as follows:

— The actual part associated with the delay value parameter Dly shall be a timing generic representing an internal signal delay or an internal clock delay.

— The actual part associated with the input signal parameter InSig shall be a static name denoting either an input port or the corresponding wire delayed signal (if it exists).

— The actual part associated with the output signal parameter OutSig shall be an internal signal. The internal signal shall have the same subtype as the signal associated with the input signal parameter.

6.4.3 VITAL process statement

A VITAL process is a key building block of a VITAL Level 1 architecture. It is a mechanism for modeling timing constraints, functionality, and path delays.

```
VITAL_process_statement ::=
    [ process_label : ]
    process ( sensitivity_list )
        VITAL_process_declarative_part
    begin
        VITAL_process_statement_part
    end process [ process_label ] ;
```

A VITAL process statement shall have a sensitivity list. The sensitivity list shall contain the longest static prefix of every signal name that appears as a primary in a context in which the value of the signal is read. These are the only signal names that the sensitivity list may contain.

6.4.3.1 VITAL process declarative part

A VITAL process declarative part is restricted to a few kinds of declarations.

```
VITAL_process_declarative_part ::=
    { VITAL_process_declarative_item }

VITAL_process_declarative_item ::=
      constant_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | VITAL_variable_declaration
```

6.4.3.1.1 VITAL variables

VITAL variables fall into two classes that are distinguished by the manner in which their members are used. A VITAL *restricted variable* is a variable that is required to store persistent data for private use by certain procedures in the VITAL standard packages. The proper functioning of these procedures requires that the content of the variable not be modified separately by the model itself; hence, the use of a restricted variable within a VITAL Level 1 architecture is strictly limited. A variable that is not a restricted variable is called an *unrestricted variable*.

```
VITAL_variable_declaration ::=
    variable identifier_list : type_mark [ index_constraint ] [ := expression ] ;
```

The type of an unrestricted variable shall be one of the standard logic types Std_ulogic and Std_logic_vector, or the standard type Boolean.

6.4.3.1.1.1 VITAL restricted variables

A restricted variable is identified by its association with a restricted formal parameter. This single association is the only use permitted for a VITAL restricted variable.

Certain formal parameters of some procedures in the VITAL standard packages are designated as *restricted formal parameters*. They are:

GlitchData	on procedure VitalPathDelay
TimingData	on procedures VitalSetupHoldCheck, VitalRecoveryRemovalCheck
PeriodPulseData	on procedure VitalPeriodPulseCheck
PreviousDataIn	on procedure VitalStateTable

The actual part in the association of a restricted formal parameter shall be the simple name of a restricted variable.

Certain restrictions are placed on the declaration of a restricted variable. The type mark in the restricted variable declaration shall denote the type or subtype denoted by the type mark in the corresponding restricted formal parameter declaration. If the declaration of the restricted variable contains an initial value expression, then that

expression shall take one of the following forms:

— It can be the name of a constant that is declared in one of the VITAL standard packages.

— It can be a function call invoking a function that is declared in one of the VITAL standard packages. Each actual parameter part in such a function call shall be a locally static expression.

6.4.3.2 VITAL process statement part

The VITAL process statement part consists of statements that describe timing constraint checks, cell function, and path delay selection.

```
VITAL_process_statement_part ::=
    [ VITAL_timing_check_section ]
    [ VITAL_functionality_section ]
    [ VITAL_path_delay_section ]
```

These statements are grouped into three distinct sections, each of which is devoted to a particular sort of specification. A VITAL process shall include at least one of these sections.

6.4.3.2.1 Timing check section

The timing check section performs timing constraint checks through the invocation of predefined timing check procedures. Timing checks that can be performed in this section include setup/hold checks, recovery/removal checks, and period/pulsewidth checks.

```
VITAL_timing_check_section ::=
    if VITAL_timing_check_condition then
        { VITAL_timing_check_statement }
    end if ;

VITAL_timing_check_condition ::= generic_simple_name

VITAL_timing_check_statement ::= procedure_call_statement
```

The timing check condition shall be a simple name denoting a TimingChecksOn control generic that shall be declared in the entity.

A timing check statement is a procedure call statement that invokes one of the VITAL timing check procedures declared in the package VITAL_Timing: VITALSetupHoldCheck, VITALRecoveryRemovalCheck, or VITALPeriodPulseCheck. Each of these procedures performs the specified check and returns a parameter value indicating whether or not a constraint violation occurred (see 8.1 for more details). These values are considered to be the output of the timing section. A timing check section is the only context in which a call to a timing check procedure is allowed.

The actual parameter part of a timing check procedure call shall satisfy the following requirements:

— The actual part associated with a formal parameter representing a signal name shall be a locally static name.

— The actual part associated with the formal parameter HeaderMsg shall be a globally static expression.

— The actual part associated with a formal parameter of the standard type Time shall be a globally static expression.

— The actual part associated with a formal parameter XOn or MsgOn shall be a locally static expression or a simple name denoting the control generic of the same name.

— A function call or operator in an actual part shall invoke a function or operator that is defined in package Standard, package Std_Logic_1164, or package VITAL_Timing.

— An actual part associated with the formal parameter TestSignalName, RefSignalName, or RefTransition shall be a locally static expression.

Timing checks shall be independent of one another. It is an error for a variable that is associated with a timing check violation parameter to appear in another timing check statement.

NOTE—Although the variable associated with the violation parameter in a timing check cannot be used in another timing check, it may be used in other sections of the VITAL process—during functionality computation, for instance.

6.4.3.2.2 Functionality section

The functionality section describes the behavior of the cell.

```
VITAL_functionality_section ::=
    { VITAL_variable_assignment_statement | procedure_call_statement }

VITAL_variable_assignment_statement ::=
    VITAL_target := expression ;

VITAL_target ::= unrestricted_variable_name
```

The function of a model is specified in terms of variable assignment statements and procedure call statements.

A procedure call statement in the functionality section shall invoke the predefined procedure VITALStateTable that is defined in package VITAL_Primitives (see 7.3.4). The actual parameter part of the procedure call shall satisfy the following requirements:

— The actual part associated with the StateTable parameter shall adhere to the restrictions relating to the StateTable parameter as described in 6.4.4.

— The constraint on the variable associated with the PreviousDataIn parameter shall match that on the actual associated with the DataIn parameter.

Certain restrictions are placed on a VITAL variable assignment statement. The target shall be an unrestricted variable that is denoted by a locally static name, and the right-hand side expression shall be such that every primary in the right-hand side expression is one of the following:

a) A globally static expression

b) A name denoting a variable, a port, or an internal signal

c) A function call invoking a standard logic function, a VITAL primitive, or the function VITALTruthTable

d) An aggregate, or a qualified expression whose operand is an aggregate

e) A parenthesized expression

The functionality section is the only context in which a call to the function VITALTruthTable (see 7.3.3) is allowed. The actual part associated with the formal parameter TruthTable in such a call shall adhere to the restrictions relating to the TruthTable parameter as described in 6.4.4.

NOTE—The function form of VITALTruthTable can be invoked only from the functionality section; however, the procedure form of VITALTruthTable can be invoked as a VITAL primitive concurrent procedure call.

6.4.3.2.3 Path delay section

The path delay section drives ports or internal signals using appropriate delay values, with provisions for glitch handling, message reporting control, and output strength mapping.

Path delay selection is modeled with a procedure call statement that invokes one of the path delay procedures—VITALPathDelay, VITALPathDelay01, or VITALPathDelay01Z—defined in the package VITAL_Timing. A path delay procedure selects the appropriate propagation delay path and schedules a new output value for the specified signal (see 9.4 for more detail). A path delay section is the only context in which a call to a path delay procedure is allowed.

The actual parameter part of a path delay procedure call shall satisfy the following requirements:

— The actual part associated with the formal parameter OutSignal shall be a locally static name.

— The actual part associated with the formal parameter Paths shall be an aggregate. Each element expression of the array aggregate shall be a record aggregate. The expression associated with a PathDelay subelement shall be globally static. The expression associated with an InputChangeTime subelement shall be either a Last_Event attribute or a locally static expression.

— The actual part associated with the formal parameter GlitchMode shall be a locally static expression.

— The actual part associated with the formal parameter GlitchData shall be a locally static name.

— The actual part associated with a formal parameter XOn or MsgOn shall be a locally static expression or a simple name denoting the control generic of the same name.

— An actual part associated with the formal parameter OutSignalName, DefaultDelay, or OutputMap shall be a locally static expression.

NOTE—Each port of mode OUT, INOUT, or BUFFER that has a driver is driven by a call to a VITAL primitive procedure or a call to a path delay procedure.

6.4.4 VITAL primitive concurrent procedure call

VITAL_primitive_concurrent_procedure_call ::=
*VITAL_primitive*_concurrent_procedure_call

A VITAL primitive concurrent procedure call provides a distributed delay modeling capability. It invokes any one of the primitives defined in the package VITAL_Primitives to compute functionality and schedule signal values using delay values selected within the procedure. A complete list of the available primitives is given in clause 7.

The actual parameter part of a primitive subprogram call shall satisfy the following requirements:

— An actual part associated with a formal parameter of class VARIABLE or SIGNAL shall be a static name.

— An actual part associated with a formal parameter of class CONSTANT shall be a globally static expression.

— An actual part associated with the formal parameter ResultMap shall be a locally static expression.

— An actual part associated with a TruthTable or StateTable formal parameter in a call to a table primitive shall be a constant that is not a deferred constant. Furthermore, the value expression of that constant shall be a positional aggregate formed using only locally static expressions or nested aggregates of this form.

7. Predefined primitives and tables

The VITAL_Primitives standard package defines a number of primitive functions and procedures that provide basic functional support for VITAL Level 1 models.

The set of VITAL primitives consists of logic primitives and utility primitives. Logic primitives perform basic logic operations. Utility primitives support multiple driver resolution and table operations.

7.1 VITAL logic primitives

Each logic primitive is defined in both function and procedure form for use in the functionality section of a VITAL process or in a VITAL primitive concurrent procedure call, respectively.

The logic primitives are given in table 2.

Table 2—VITAL logic primitives

VitalAND	VitalAND2	VitalAND3	VitalAND4
VitalOR	VitalOR2	VitalOR3	VitalOR4
VitalXOR	VitalXOR2	VitalXOR3	VitalXOR4
VitalNAND	VitalNAND2	VitalNAND3	VitalNAND4
VitalNOR	VitalNOR2	VitalNOR3	VitalNOR4
VitalXNOR	VitalXNOR2	VitalXNOR3	VitalXNOR4
VitalBUF	VitalBufIf0	VitalBufIf1	VitalIDENT
VitalINV	VitalInvIf0	VitalInvIf1	–
VitalMux	VitalMux2	VitalMux3	VitalMux4
VitalDecoder	VitalDecoder2	VitalDecoder4	VitalDecoder8

7.1.1 Logic primitive functions

VITAL logic primitive functions compute the defined function and return a value of type Std_Ulogic or Std_Logic_Vector. All parameters of the logic primitive functions are constants of mode IN.

Example:

```
ARCHITECTURE PinToPinDelay of AndOr IS
    attribute VITAL_LEVEL1 of PinToPinDelay: architecture is TRUE;
BEGIN
    VitalBehavior: PROCESS (A, B, C, D)
```

```
        VARIABLE AND1_Out, AND2_Out, Q_Zd: std_ulogic;
        VARIABLE GlitchData_Q : VitalGlitchDataType;
    BEGIN
        -- Functionality section
        AND1_Out  := VitalAND2 ( A, B );
        AND2_Out  := VitalAND2 ( C, D );
        Q_zd    := VitalOR2  (AND1_Out, AND2_Out, ResultMap => DefaultECLMap);
        -- Path delay section
        VitalPathDelay01 (
            OutSignal => Q,
            OutSignalName => "Q",
            OutTemp => Q_zd,
            Paths => (
                0 => (InputChangeTime => A'last_event,
                      PathDelay => tpd_A_Q,
                      PathCondition => TRUE),
                1 => (InputChangeTime => B'last_event,
                      PathDelay => tpd_B_Q,
                      PathCondition => TRUE),
                2 => (InputChangeTime => C'last_event,
                      PathDelay => tpd_C_Q,
                      PathCondition => TRUE),
                3 => (InputChangeTime => D'last_event,
                      PathDelay => tpd_D_Q,
                      PathCondition => TRUE)
            ),
            GlitchData => GlitchData_Q,
            DefaultDelay => VitalZeroDelay01,
            Mode => OnDetect,
            XoN => TRUE,
            MsgOn => TRUE,
            MsgSeverity => WARNING);
    END PROCESS;
END;
```

7.1.2 Logic primitive procedures

VITAL logic primitive procedures execute in a manner similar to that of a separate process. The procedures wait internally for an event on an input signal, compute the new result, perform glitch handling, schedule transactions on the output signals, and wait for further input events. All of the functional (logic) input or output parameters of the primitive procedures are signals. All other parameters are constants.

The procedure primitives are parameterized for separate path delays from each input signal. All path delays default to 0 ns.

Example:

```
ARCHITECTURE DistributedDelay OF AndOr IS
    ATTRIBUTE VITAL_LEVEL1 of DistributedDelay: architecture IS TRUE;
    SIGNAL AND1_Out, AND2_Out: std_ulogic;
BEGIN
    I1: VitalAND2 (AND1_Out, A, B, tdevice_I1_Q, tdevice_I1_Q);
    I2: VitalAND2 (AND2_Out, C, D, tdevice_I2_Q, tdevice_I2_Q);
    I3: VitalOR2   (Q, AND1_Out, AND2_Out, tdevice_I3_Q, tdevice_I3_Q);
```

```
END;
```

7.1.3 Establishing output strengths

Each logic primitive function or procedure by default produces an output value from the set of values {'U', 'X', '0', '1', 'Z'}. This set of logic strengths may be expanded through use of the optional ResultMap parameter (of type ResultMapType), which provides rapid conversion of any of the standard five output values to any other output value through the use of a simple table lookup.

```
type ResultMapType is array (UX01) of Std_ulogic;
type ResultZMapType is array (UX01Z) of Std_ulogic;

constant VitalDefaultResultMap : VitalResultMapType := ( 'U', 'X', '0', '1' );
constant VitalDefaultResultZMap : VitalResultZMapType := ( 'U', 'X', '0', '1', 'Z' );
```

Example:

```
ARCHITECTURE Structural OF Pullup IS
    ATTRIBUTE VITAL_LEVEL1 of Structural: architecture IS TRUE;
    CONSTANT DefaultECLMap : VitalResultMapType := ('U', 'X', 'L', '1');
BEGIN
    I1: VitalBUF (Q, A, tpd_A_Q, ResultMap => DefaultECLMap);
END;
```

In this example, the constant DefaultECLMap that is supplied as the ResultMap actual parameter causes a '0' to be mapped to 'L' on the output of the primitive.

7.2 VitalResolve

The procedure VitalResolve supports the resolution of multiple signal drivers, allowing a model to drive these multiple signals on a single signal. It invokes the standard logic function Resolved on the input vector and assigns it to the outputs with zero delay.

Example:

```
ARCHITECTURE Structural OF ResolvedLogic IS
    ATTRIBUTE VITAL_LEVEL1 of Structural: architecture IS TRUE;
    SIGNAL Q_Temp1, Q_Temp2 : std_ulogic;
BEGIN
    I1: VitalAND2 (Q_Temp1, A, B, tdevice_I1_Q, tdevice_I1_Q);
    I2: VitalAND2 (Q_Temp2, C, D, tdevice_I2_Q, tdevice_I2_Q);
    R1: VitalResolve (Q, (Q_Temp1, Q_Temp2));
END;
```

7.3 VITAL table primitives

The VITAL_Primitives package supports the standard specification and use of truth tables and symbol tables through the use of the table primitives VitalTruthTable and VitalStateTable. VitalTruthTable is provided for modeling combinational cells. VitalStateTable is provided for modeling sequential cells.

7.3.1 VITAL table symbols

A transition set or a steady-state condition is represented by a special table symbol. The symbol set defined by the

type VitalTableSymbolType is used to specify high-accuracy state tables.

```
type VitalTableSymbolType is (
    '/',    -- 0 -> 1
    '\',    -- 1 -> 0
    'P',    -- Union of '/' and '^' (any edge to 1)
    'N',    -- Union of '\' and 'v' (any edge to 0)
    'r',    -- 0 -> X
    'f',    -- 1 -> X
    'p',    -- Union of '/' and 'r' (any edge from 0)
    'n',    -- Union of '\' and 'f' (any edge from 1)
    'R',    -- Union of '^' and 'p' (any possible rising edge)
    'F',    -- Union of 'v' and 'n' (any possible falling edge)
    '^',    -- X -> 1
    'v',    -- X -> 0
    'E',    -- Union of 'v' and '^' (any edge from 'X')
    'A',    -- Union of 'r' and '^' (rising edge to or from 'X')
    'D',    -- Union of 'f' and 'v' (falling edge to or from 'X')
    '*',    -- Union of 'R' and 'F' (any edge)
    'X',    -- Unknown level
    '0',    -- low level
    '1',    -- high level
    '-',    -- don't care
    'B',    -- 0 or 1
    'Z',    -- High Impedance
    'S'     -- steady value
  );
```

The acceptable range of table symbols for each sort of table is defined with a scalar subtype definition. A truth or state table can be constructed from any value in the subset of values defined by the corresponding table symbol subtype.

```
subtype VitalTruthSymbolType is VitalTableSymbolType range 'X' to 'Z';
subtype VitalStateSymbolType is VitalTableSymbolType range '/' to 'S';
```

Table 3 shows the VitalTableSymbolType elements and the levels and edge transitions that they represent.

A truth or state table is partitioned into different sections, each of which represents a specific kind of information. These sections include an *input pattern* and a *response*. For a state table, an additional *state* section is included. The input pattern section shall not contain the symbol 'Z'. The response section shall contain only the symbols 'X', '0', '1', '-' and 'Z', and for a state table, the symbol 'S as well. The state section of a state table can only contain the symbols 'X', '0', '1', '-' and 'B'. It is an error if any symbols other than those allowed are encountered in a section.

NOTES

1—The table symbols are enumeration literals, therefore they are case-sensitive.

2—A limited set of table symbol values can be used to develop truth tables. Any table symbol can be used in a state table.

7.3.2 Table symbol matching

During truth or state table processing, the input to the table primitive (DataIn) is matched to the stimulus portion of the table. The matching process begins by converting the input data to the equivalent 'X', '0', or '1' values by

Table 3—Truth Table and State Table symbol semantics

VITAL table symbols	Level and edge transitions								
	00	10	X0	11	01	X1	0X	1X	XX
'/'					*				
'\'		*							
'P'					*	*			
'N'		*	*						
'r'							*		
'f'								*	
'p'					*		*		
'n'		*						*	
'R'					*	*	*		
'F'		*	*					*	
'^'						*			
'v'			*						
'E'			*			*			
'A'						*	*		
'D'			*					*	
'*'		*	*		*	*	*	*	
'X'							*	*	*
'0'	*	*	*						
'1'				*	*	*			
'-'	*	*	*	*	*	*	*	*	*
'B'	*	*	*	*	*	*			
'Z'									
'S'	*			*					

applying the standard logic TO_X01 function. The resulting values are then compared to the stimulus portion of the table according to the matching rules in table 4.

For a state table, the current and previous values of DataIn are used to determine if an edge has occurred. These edges are matched with the edge entries that are specified in the input pattern of the table using the semantics of the edge symbols shown in table 3.

7.3.3 TruthTable primitive

A function version of VitalTruthTable is defined for use inside a VITAL process. The procedure version of

Table 4—Matching of table symbols to input stimulus

Table stimulus portion	DataInX01 := To_X01(DataIn)	Result of comparison
'X'	'X'	'X' only matches with 'X'
'0'	'0'	'0' only matches with '0'
'1'	'1'	'1' only matches with '1'
'-'	'X', '0', '1'	'-' matches with any value of DataInX01
'B'	'0', '1'	'B' only matches with '0' or '1'

VitalTruthTable is defined for use in a concurrent procedure call. In addition to performing the same result computation as the function version, the procedure version schedules the resulting value on the output signal with a delay of 0 ns. Overloaded forms are provided to support both scalar and vector output.

7.3.3.1 Truth table construction

A VITAL truth table is an object of type VitalTruthTableType.

```
type VitalTruthTableType is array ( Natural range <>, Natural range <>) of VitalTruthSymbolType;
```

The length of the first dimension of a truth table is the number of input combinations that have a specified output value. The length of the second dimension shall be the sum of the size of the input pattern section and the size of response section.

The number of inputs to the truth table shall be equal to the length of the DataIn parameter. It is an error if the length of DataIn is greater than or equal to the size of the second dimension of the TruthTable parameter.

A row in a truth table consists of two sections: an *input pattern* and a *response*. A row *i* of the truth table is interpreted as follows:

```
InputPattern(j downto 0), Response(k downto 0)
where
    j = DataIn'Length - 1
    k = TruthTable'Length(2) - DataIn'Length - 1
```

Example:

Truth table for a 2 to 4 decoder:

```
Constant DecoderTable: VitalTruthTableType(0 to 3, 0 to 5) :=
--  Input Pattern       Response
--  D1      D0          Q3  Q2  Q1  Q0

((  '0',    '0',        '0', '0', '0', '1'),
 (  '0',    '1',        '0', '0', '1', '0'),
 (  '1',    '0',        '0', '1', '0', '0'),
 (  '1',    '1',        '1', '0', '0', '0'));
```

7.3.3.2 TruthTable algorithm

The VitalTruthTable primitive compares the stimulus, DataIn, with the input pattern section of each row (starting from the top) in TruthTable to find the first matching entry. If all of the subelements of DataIn match with corresponding subelements of the input pattern of a particular row in TruthTable, the outputs are determined from the response section of the corresponding row. The outputs are then converted to the standard logic X01Z subtype. If all rows in TruthTable are searched and no match is found, then VitalTruthTable returns either an 'X' or a vector of 'X's, as appropriate.

The vector form of the procedure places the outputs in the actual associated with the parameter Result, starting from the right side of both the truth table and the actual associated with Result, until the actual is filled or there are no more outputs left in the truth table. It is an error if Result is too small or too large to hold all of the values. The vector function behaves in a manner similar to the vector procedure; however, it always returns a vector with the range TruthTable'Length(2) - DataIn'Length - 1 downto 0.

7.3.4 StateTable primitive

There are two versions of the VitalStateTable procedure—one that is intended for use as a sequential statement and one that is intended for use as a concurrent statement. The concurrent statement version of this procedure performs the same result computation as the function version, but in addition it schedules the resulting value on the output signal with a delay of 0 ns. Overloaded forms are provided to support both scalar and vector output.

7.3.4.1 State table construction

A VITAL state table is an object of type VitalStateTableType.

```
type VitalStateTableType is array ( Natural range <>, Natural range <> ) of VitalStateSymbolType;
```

The length of the first dimension of a state table is the number of input and state combinations that have a specified output value. The length of the second dimension is the sum of the length of the input patterns section, the length of the state section, and the length of the response section.

The number of inputs to the state table shall equal the length of the DataIn parameter. It is an error if the length of DataIn is greater than or equal to the size of the second dimension of the StateTable parameter.

A row in a state table consists of the following sections: an *input pattern*, a *state*, and a *response*. Each row in the table shall have at most one element from the subtype VitalEdgeSymbolType. Each row *i* of the StateTable is interpreted as follows:

InputPattern(*j* **downto** 0), State(*k* **downto** 0), Response(*l* **downto** 0)
where
j = DataIn'Length - 1
k = NumState - 1
l = StateTable'Length(2) - DataIn'Length - NumState -1

NOTE: A state table should include at least one entry with an 'S' for the clock so that VitalStateTable can handle the case in which the procedure is activated but the clock did not change. If this entry is not included, then the result defaults to 'X's.

Example:

State table for a positive-edge triggered D Flip flop:

```
Constant DFFTable: VitalStateTableType :=
--  RESET     D       CLK     State   Q
```

```
((  '0',    '-',    '-',    '-',    '0'),
(   '1',    '1',    '/',    '-',    '1'),
(   '1',    '0',    '/',    '-',    '0'),
(   '1',    'X',    '/',    '-',    'X'),
(   '1',    '-',    '-',    '-',    'S'));
```

7.3.4.2 StateTable algorithm

The procedure VitalStateTable computes the value of the output of a synchronous sequential circuit (a Moore machine) based on the inputs, the present state, and a state table. These procedures compare the stimulus, DataIn (and edges on it), with the input pattern section of each row (starting from the top) in StateTable to find the first matching entry. If all input entries are found to match, the comparison moves to the states. Here the comparison moves from the leftmost index of Result [comparing it to State(NumStates - 1) in the state table] and proceeds to the right. The comparison of the entry continues until all of the inputs have been compared or a mismatch is encountered. The search terminates with the first level or edge match or when the table entries are exhausted. If all rows in StateTable are searched and no match is found, then the actual associated with the formal parameter Result is assigned an 'X' or a vector of 'X's, as appropriate.

Once a match is found, or it is determined that no match can be made, the new values of the state variables and the outputs are determined from the response section of the state table. The states and outputs are placed into the parameter Result, starting from the right side of both the state table and Result, until Result is filled or there are no more outputs or states left in the state table. It is an error if Result is too small or too large to hold all of the values.

8. Timing constraints

This standard provides support for standard timing constraint checking and for the modeling of negative timing constraints.

8.1 Timing check procedures

The package VITAL_Timing defines three kinds of timing check procedures: VitalSetupHoldCheck, VITALRecoveryRemovalCheck, and VITALPeriodPulseCheck. Each is overloaded for use with test signals of type Std_Ulogic or Std_Logic_Vector. Each defines a CheckEnabled parameter that supports the modeling of conditional timing checks.

A VITAL timing check procedure performs the following functions:

— It detects a timing constraint violation if the timing check is enabled.

— It reports a timing constraint violation using a VHDL assertion statement. The report message and severity level of the assertion are controlled by the model.

— It sets the value of a corresponding violation flag. If a timing violation is detected, the value of this flag is set to 'X'; otherwise, it is set to '0'. 'X' generation for this flag can be controlled by the model.

The same timing check procedures are used for both positive and negative timing constraint values. Two delay parameters—TestDelay and RefDelay—are defined for modeling the delays associated with the test or reference signals when negative setup or hold constraints are in effect. The delay parameters shall have the value zero when negative constraints do not apply.

8.1.1 VitalSetupHoldCheck

The procedure VitalSetupHoldCheck detects the presence of a setup or hold violation on the input test signal with respect to the corresponding input reference signal. The timing constraints are specified through parameters representing the high and low values for the setup and hold times. This procedure assumes nonnegative values for setup/hold timing constraints.

Setup/hold constraint checks are performed by this procedure only if the CheckEnabled condition evaluates to True; however, event times required for constraint checking are always updated, regardless of the value of CheckEnabled. Setup constraints are checked in the simulation cycle in which the reference edge occurs. A setup violation is detected if the time since the last TestSignal change is less than the expected setup constraint time. Hold constraints are checked in the simulation cycle in which an event on TestSignal occurs. A hold violation is detected if the time since the last reference edge is less than the expected hold constraint time.

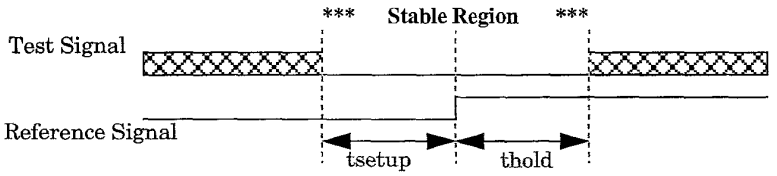


Figure 2—Setup/Hold check for positive constraints

8.1.2 VitalPeriodPulseCheck

The procedure VitalPeriodPulseCheck checks for minimum and maximum periodicity and pulse width for '1' and '0' values of the input test signal. The timing constraint is specified through parameters representing the minimal period between successive rising or falling edges of the input test signal, and the minimum pulse widths associated with high and low values.

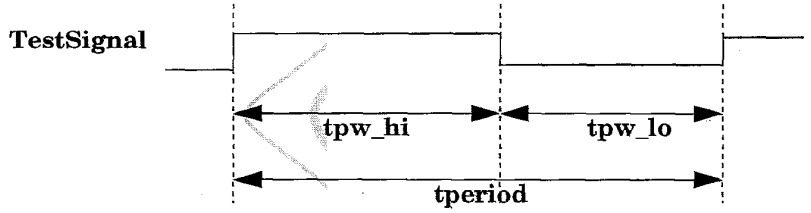


Figure 3—Period/Pulsewidth check

8.1.3 VitalRecoveryRemovalCheck

The procedure VitalRecoveryRemovalCheck detects the presence of a recovery or removal violation on the input test signal with respect to the corresponding input reference signal. It assumes nonnegative values of recovery/removal timing constraints. The timing constraint is specified through parameters representing the recovery and removal times associated with a reference edge of the reference signal. A flag indicates whether a test signal is asserted when it is high or when it is low.

Recovery/removal constraint checks are performed by this procedure only if the CheckEnabled condition evaluates to True; however, event times required for constraint checking are always updated, regardless of the value of CheckEnabled. Recovery constraints are checked in the simulation cycle in which the reference edge occurs. A recovery violation is detected if the time since the last TestSignal change is less than the expected recovery constraint time. Removal constraints are checked in the simulation cycle in which an event on TestSignal occurs. A removal violation is detected if the time since the last reference edge is less than the expected removal constraint time.

Example:

```
VITALBehavior: PROCESS (D, CLK, RESET)
    VARIABLE SetupHoldInfo : VitalTimingDataType := VitalTimingDataInit ; --Restricted variable
    VARIABLE PeriodDataInfo : VitalPeriodDataType := VitalPeriodDataInit ; --Restricted variable
```

```
    VARIABLE RecoRemoInfo : VitalTimingDataType := VitalTimingDataInit ; --Restricted variable
    VARIABLE Violation_flag, Viol1, Viol2, Viol3 : X01;
    ...
BEGIN
    -- Timing Check Section
    IF (TimingChecksOn) THEN
        -- Setup/hold check between D and rising CLK
        VitalSetupHoldCheck (
            TestSignal     => D,              TestSignalName   => "D",
            RefSignal      => CLK,            RefSignalName    => "CLK",
            SetupHigh      => tsetup_D_CLK,   SetupLow         => tsetup_D_CLK,
            HoldHigh       => thold_D_CLK,    HoldLow          => thold_D_CLK,
            CheckEnabled   => RESET = '1',    RefTransition    => 'R',
            MsgOn          => TRUE,           XOn              => TRUE,
            HeaderMsg      => "Instance1",    TimingData       => SetupHoldInfo,
            Violation      => Viol1,          MsgSeverity      => ERROR);
        -- Pulsewidth and period check for CLK
        VitalPeriodPulseCheck (
            TestSignal     => CLK,            TestSignalName   => "CLK",
            Period         => tperiod_CLK,
            PulseWidthHigh => tpw_CLK_posedge,
            PulseWidthLow  => tpw_CLK_negedge,
            PeriodData     => PeriodDataInfo, Violation        => Viol2,
            MsgOn          => TRUE,           XOn              => TRUE,
            HeaderMsg      => "Instance1",
            CheckEnabled   => RESET = '1',    MsgSeverity      => ERROR);
        -- Recovery/removal check between RESET and rising CLK
        VitalRecoveryRemovalCheck (
            TestSignal     => RESET,          TestSignalName   => "RESET",
            RefSignal      => CLK,            RefSignalName    => "CLK",
            Recovery       => trecovery_RESET_CLK,
            Removal        => tremoval_RESET_CLK,
            ActiveLow      => FALSE,          CheckEnabled     => RESET = '1',
            RefTransition  => 'R',
            MsgOn          => TRUE,           XOn              => TRUE,
            HeaderMsg      => "Instance1",    TimingData       => RecoRemoInfo,
            Violation      => Viol3,          MsgSeverity      => ERROR);
    END IF;
    Violation_flag := Viol1 or Viol2 or Viol3;
    ...
END PROCESS;
```

8.2 Modeling negative timing constraints

Some devices may be characterized with negative setup or hold times, or negative recovery or removal times. If any of these values is negative, then the data constraint interval does not overlap the reference clock edge, and a *negative timing constraint* is said to be *in effect*.

A negative hold or removal time corresponds to an internal delay on the test (or data) signal. A negative setup or recovery time corresponds to an internal delay on the reference (or clock) signal. These internal delays determine when a data signal is sampled on the edge of the clock signal. Special adjustments are required in the case of negative timing constraints because the data value at the time that the clock edge is detected may be different from the data value during the constraint interval. Furthermore, the setup time may be difficult to check because a

violating data edge may not be the most recent data edge preceding the clock.

Negative timing constraints in a VITAL Level 1 model are handled by internally delaying the test or reference signals. Negative setup or recovery times result in a delayed reference signal. Negative hold or removal times result in a delayed test signal. Furthermore, the delays associated with other signals may need to be appropriately adjusted so that all constraint intervals overlap the delayed reference signals. After these delay adjustments are performed, the timing constraint values on the timing check procedures are always nonnegative.

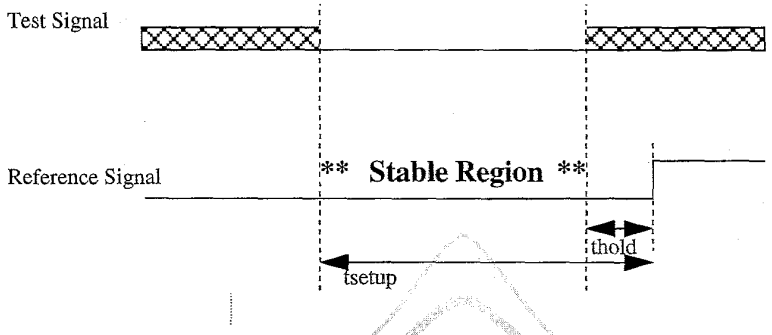


Figure 4—The data constraint interval for a negative hold constraint

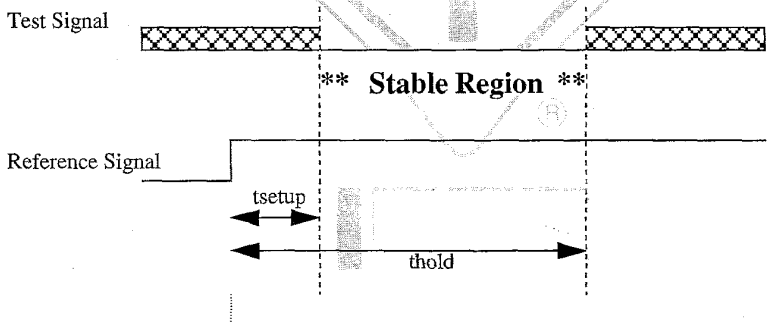


Figure 5—The data constraint interval for a negative setup constraint

8.2.1 Requirements on the VHDL description

This standard defines a delay adjustment algorithm that transforms negative delay values to nonnegative values. This algorithm relies on certain model characteristics in order to calculate the delays correctly; therefore, a negative timing constraint has to be anticipated and the model structured to accommodate it.

To model negative constraints in a VITAL-compliant model, the corresponding VHDL description shall contain the following:

— The declaration of an internal clock delay generic for each clock (reference) signal that is associated with a negative setup (or recovery) constraint

— The declaration of an internal signal delay generic for every data (test) signal that is associated with a negative hold (or removal) constraint

— The declaration of biased propagation delay generics for paths that are dependent on multiple clocks

— A signal delay block in the architecture that contains calls to the VitalSignalDelay procedure to delay the appropriate test/reference signal

These rules are part of the VITAL Level 0 modeling specification (see clause 4).

NOTE—In general, a model should read the value of the internally delayed signal rather than that of the corresponding signal that is not delayed; however, the model is not prohibited from reading the value of the signal that is not delayed.

8.2.2 Negative constraint calculation phase

The negative constraint delay adjustments are computed outside of the actual VHDL model in a special phase of simulation called the *negative constraint calculation phase*, which occurs directly after the VITAL backannotation phase and directly before normal VHDL initialization.

Negative constraint calculation is performed for each VITAL Level 0 instance that defines a negative constraint timing generic. The values of certain timing generics are computed and set, and the values of others are adjusted in an iterative algorithm that uses the generic values set during previous steps.

Negative constraint calculation is performed in the following sequence:

a) Calculate internal clock delays

b) Calculate internal signal delays

c) Calculate biased propagation delays

d) Adjust propagation delays

e) Adjust timing constraint values corresponding to setup, hold, recovery, and removal times

It is an error if at the end of the negative constraint calculation stage, a timing generic that is adjusted by this algorithm still has a negative value.

NOTE—A calculation or adjustment that is performed as a part of the negative constraint calculation phase may result in a reduction in the value of a generic (or one of its subelements) that causes the value to become negative, in which case the negative constraint algorithm replaces the negative value with a zero value. This situation may or may not indicate an error; hence, a tool that processes VITAL-compliant models may choose to issue a warning when it replaces the negative value.

8.2.2.1 Calculation of internal clock delays

The value of each internal clock delay generic is computed as follows:

a) The name of the associated clock signal is extracted from the <ClockPort> portion of the internal clock delay generic name.

b) All setup and recovery timing generics on the same instance are examined. Those generics for which the <ReferencePort> part of the generic name is the same as the <ClockPort> name are marked.

c) The minimum value of all the subelements of all the marked timing generics is determined. If that value is negative, the internal clock delay generic receives the absolute value; otherwise, it is set to 0 ns.

8.2.2.2 Calculation of internal signal delays

The value of each internal signal delay generic is computed as follows:

a) The names of the associated clock and input signals are extracted from the <ClockPort> and <InputPort> portions of the internal signal delay generic name.

b) If there is an internal clock delay generic containing the same clock signal name, then its value is the associated clock delay. Otherwise, the associated clock delay is 0 ns.

c) All hold and removal timing generics on the same instance are examined. Those generics for which the <ReferencePort> part of the generic name is the same as the <ClockPort> name and the <TestPort> part of the generic name is the same as the <InputPort> name are marked.

d) The minimum value of all subelements of all the marked timing generics is determined. This value is reduced by the associated clock delay. If the resulting value is negative, it is replaced by its absolute value; otherwise, it is replaced by 0 ns.

8.2.2.3 Calculation of biased propagation delays

The value of each biased propagation delay generic is computed as follows:

a) The corresponding propagation delay generic (denoting the same input and output ports, condition name, and edge) is identified (see 4.3.2.1.3.14). The value of the biased propagation delay generic is initialized to the value of the corresponding propagation delay generic.

b) The names of the associated clock and input signals are extracted from the <ClockPort> and <InputPort> portions of the biased propagation delay generic name.

c) If there is an internal signal delay generic (see 4.3.2.1.3.13) on the same instance whose name denotes the same <InputPort> and <ClockPort> parts, then the value of each subelement of the biased propagation delay generic is reduced by the value of that internal signal delay generic. If the resulting value of any subelement is negative, then the value of that subelement is set to zero.

NOTE—Due to the name construction of the internal signal delay generic, there can be only one internal signal delay generic that matches both the InputPort and ClockPort names (in step c).

8.2.2.4 Adjustment of propagation delay values

Propagation delay generics are adjusted in two separate steps:

a) All propagation delay timing generics from a clock signal are adjusted

b) Propagation delays that do not correspond to a biased propagation delay generic are adjusted.

It is an error if a propagation delay generic is adjusted by more than one internal signal delay.

8.2.2.4.1 Adjustment of clock to output propagation delay values

Each internal clock delay generic is adjusted as follows:

a) The name of the associated clock signal is extracted from the <ClockPort> portion of the internal clock delay generic name.

b) All propagation delay generics on the same instance are examined. Those generics for which the <InputPort> part of the generic name is the same as the <ClockPort> name are marked.

c) The value of each subelement of each marked generic is reduced by the value of the internal clock delay generic. If the resulting value of any subelement is negative, then the value of the subelement is set to 0 ns.

8.2.2.4.2 Adjustment of other propagation delay values

Each internal signal delay timing generic is adjusted as follows:

a) The names of the associated clock and input signals are extracted from the <ClockPort> and <InputPort> portions of the internal signal delay generic name.

b) All propagation delay generics on the instance are examined. If the generic was identified as corresponding to a biased propagation delay generic during the calculation of biased propagation delays, then it is not marked. Otherwise, those generics for which the <InputPort> part of the generic name is the same as the <InputPort> name are marked.

c) The value of each subelement of each marked generic is reduced by the value of the internal signal delay generic. If the resulting value of any subelement is negative, then the value of the element is set to 0 ns.

8.2.2.5 Adjustment of timing check generics

The timing check generics—setup, hold, recovery, and removal generics—are adjusted in two separate steps.

8.2.2.5.1 Internal clock delay generic

For each internal clock delay generic:

a) The name of the associated clock port is extracted from the <ClockPort> portion of the internal clock delay generic name.

b) All setup and recovery generics on the same instance are examined. Those generics for which the <ReferencePort> part of the generic name is the same as the <ClockPort> name are marked. The value of each subelement of each marked generic is increased by the value of the internal clock delay generic.

c) All hold and removal generics on the same instance are examined. Those generics for which the <ReferencePort> part of the generic name is the same as the <ClockPort> name are marked. The value of each subelement of each marked generic is reduced by the value of the internal clock delay generic. If the resulting value of any subelement is negative, then the value of the subelement is set to 0 ns.

8.2.2.5.2 Internal signal delay generic

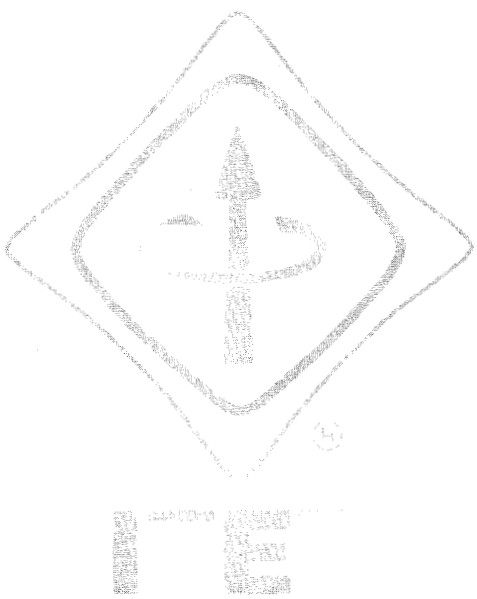
For each internal signal delay generic:

a) The names of the associated clock and input ports are extracted from the <ClockPort> and <InputPort> portions of the internal signal delay generic name.

b) All setup and recovery generics on the same instance are examined. Those generics for which the <ReferencePort> part of the generic name is the same as the <ClockPort> name and the <TestPort> part of the generic name is the same as the <InputPort> name are marked. The value of each subelement of

each marked generic is reduced by the value of the internal signal delay generic. If the resulting value of any subelement is negative, then the value of the subelement is set to 0 ns.

c) All hold and removal generics on the same instance are examined. Those generics for which the <ReferencePort> part of the generic name is the same as the <ClockPort> name and the <TestPort> part of the generic name is the same as the <InputPort> name are marked. The value of each subelement of each marked generic is increased by the value of the internal signal delay generic.



9. Delay selection

This standard supports propagation delay path selection and signal output scheduling in both sequential and concurrent contexts. These activities are performed by a number of predefined procedures provided for use by VITAL Level 1 models. The predefined procedures are as follows:

— The VITAL path delay procedures, for use in the path delay section of a VITAL process.

— The VITAL concurrent primitives, for use in concurrent procedure calls.

9.1 VITAL delay types and subtypes

The package VITAL_Timing defines a number of delay types and subtypes that support the specification and selection of simple delay values as well as delay values corresponding to the transitions between any of the states '0', '1', 'X', and 'Z'. A delay whose value may vary according to the nature of a transition is called a *transition-dependent delay*. A delay with no such dependence is a *simple delay*.

```
type VitalTransitionType is ( tr01, tr10, tr0z, trz1, tr1z, trz0, tr0x, trx1, tr1x, trx0, trxz, trzx);

subtype VitalDelayType is Time;
type VitalDelayType01 is array (VitalTransitionType range tr01 to tr10) of Time;
type VitalDelayType01Z is array (VitalTransitionType range tr01 to trz0) of Time;
type VitalDelayType01ZX is array (VitalTransitionType range tr01 to trzx) of Time;

type VitalDelayArrayType is array (NATURAL range <>) of VitalDelayType;
type VitalDelayArrayType01 is array (NATURAL range <>) of VitalDelayType01;
type VitalDelayArrayType01Z is array (NATURAL range <>) of VitalDelayType01Z;
type VitalDelayArrayType01ZX is array (NATURAL range <>) of VitalDelayType01ZX;
```

A transition-dependent delay is represented by a value of a *transition-dependent delay type*. Similarly, a simple delay is represented by a value of a *simple delay type*. There are a number of different transition-dependent delay types representing different subsets of transitions. Each kind of simple or transition-dependent delay type has both scalar and vector forms. The vector forms represent delay values corresponding to one or more vector ports for which the delay(s) associated with each bit may be different.

A value of a transition-dependent delay type associates a (possibly) different delay value with each transition in a set of transitions. The value takes the form of an array of delay times, indexed by transition values. Each element delay value is associated with the transition corresponding to its index position. The transition-dependent delay types are VitalDelayType01, VitalDelayType01Z, VitalDelayType01ZX, VitalDelayArrayType01, VitalDelayArrayType01Z, and VitalDelayArrayType01ZX. The first three are scalar forms, and the last three are vector forms.

A value of a simple delay type is a single delay value or a vector of single delay values corresponding to one or more vector ports. Although the vector form of a simple delay is an array, the delays that it represents are not associated with transitions. The simple delay types and subtypes include Time, VitalDelayType, and

VitalDelayArrayType. The first two are scalar forms, and the latter is the vector form.

The simple delay types and subtypes and the transition-dependent delay types comprise the set of *VITAL delay types and subtypes*. No other type or subtype is considered to be a VITAL delay type or subtype.

9.2 Transition-dependent delay selection

Delay selection for a particular signal may be based upon the new and previous values of the signal; this selection mechanism is called *transition-dependent delay selection*. Transitions between the previous and new values are described by enumeration values of the predefined type VitalTransitionType. Table 5 describes the delay selection for a set of previous and current values.

Table 5—Transition-dependent delay selection

Previous value	New value	Delay selected for VitalDelayType	Delay selected for VitalDelayType01	Delay selected for VitalDelayType01Z
'0'	'1'	Delay	Delay(tr01)	Delay(tr01)
'0'	'Z'	Delay	Delay(tr01)	Delay(tr0Z)
'0'	'X'	Delay	Delay(tr01)	Min(Delay(tr01), Delay(tr0Z))
'1'	'0'	Delay	Delay(tr10)	Delay(tr10)
'1'	'Z'	Delay	Delay(tr10)	Delay(tr1Z)
'1'	'X'	Delay	Delay(tr10)	Min(Delay(tr10), Delay(tr1Z))
'Z'	'0'	Delay	Delay(tr10)	Delay(trZ0)
'Z'	'1'	Delay	Delay(tr01)	Delay(trZ1)
'Z'	'X'	Delay	Min(Delay(tr10), Delay(tr01))	Min(Delay(trZ1), Delay(trZ0))
'X'	'0'	Delay	Delay(tr10)	Max(Delay(tr10), Delay(trZ0))
'X'	'1'	Delay	Delay(tr01)	Max(Delay(tr01), Delay(trZ1))
'X'	'Z'	Delay	Max(Delay(tr10), Delay(tr01))	Max(Delay(tr1Z), Delay(tr0Z))

9.3 Glitch handling

A *glitch* occurs when a new transaction is scheduled to occur at an absolute time that is greater than the absolute time of a previously scheduled pending event. Glitch handling in a VITAL Level 1 model is incorporated into the signal scheduling mechanism.

This standard supports four modes of signal scheduling. These modes are represented by the enumeration values of the predefined VITAL type VitalGlitchKindType:

```
type VitalGlitchKindType is (OnEvent, OnDetect, VitalInertial, VitalTransport);
```

The VitalInertial and VitalTransport modes are identical to the inertial and transport modes of VHDL. The OnEvent and OnDetect modes are special modes for glitch handling. In the OnEvent mode, a glitch causes an 'X' value to be scheduled on the output at the time when the scheduled event was to occur. In the OnDetect mode, a glitch causes an 'X' value to be scheduled on the output at the time of glitch detection.

Example:

Consider a simple buffer experiencing a delay. The outputs across the buffer corresponding to various glitch modes are shown in figure 6.

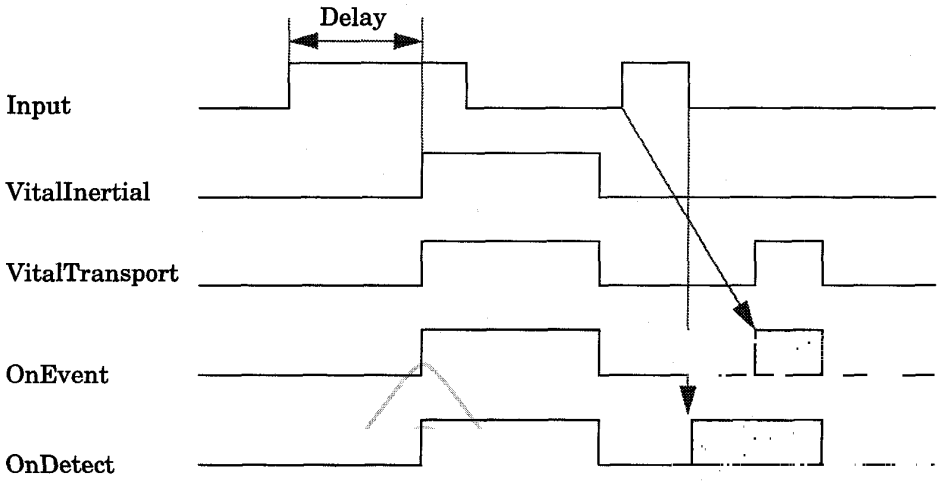


Figure 6—VITAL delay modes

9.4 Path delay procedures

Signal scheduling inside a VITAL Level 1 process can be performed by one of the predefined path delay procedures VitalPathDelay, VitalPathDelay01, and VitalPathDelay01Z. Each of these procedures provides the following capabilities:

— Transition-dependent path delay selection

— User-controlled glitch detection, 'X' generation, and violation reporting

— Scheduling of the computed values on the specified signal

The information about all the relevant paths to a particular output is specified by using the Paths parameter. The following record structure is used to convey information about an input to output path:

```
type VitalPath01Type is record
    InputChangeTime     : TIME;              -- Time stamp for path input signal
    PathDelay           : VitalDelayType01;  -- Delay for this path
    PathCondition       : BOOLEAN;           -- Path sensitize condition
end record;
```

Selection of the appropriate path delay begins with the selection of candidate paths. The candidate paths are selected by identifying the paths for which the PathCondition is true. If there is a single candidate path, then its delay is the one selected. If there is more than one candidate path, then the shortest delay (accounting for the InputChangeTime) is selected using transition-dependent delay selection. If there are no candidate paths, then the delay specified by the DefaultDelay parameter to the path delay procedure is used.

9.4.1 VitalPathDelay andVitalPathDelay01

The VitalPathDelay and VitalPathDelay01 procedures schedule path delays on signals for which the transition to 'Z' is not important. These procedures are distinguished from one another by the type of delay values that they accept. The procedure VitalPathDelay is defined for simple path delays of type VitalDelayType. Procedure VitalPathDelay01 is defined for transition-dependent path delays of type VitalDelayType01 (rise/fall delays).

Example:

```
VitalPathDelay01(
    OutSignal       => QN,     -- Signal being scheduled
    OutSignalName => "QN",                -- Name of the signal
    OutTemp         => QN_zd,             -- New signal value to be scheduled
      Paths           => (                -- One path data for each input
                                          -- affecting the output
    -- First input pin that affects the output
      0   => (InputChangeTime => CLK_ipd'LAST_EVENT,
              PathDelay       => tpd_CLK_QN,
              Condition       => (PN_ipd = '0' and CN_ipd = '1' )),
    -- Second input pin that affects the output
      1   => (InputChangeTime => PN_ipd'LAST_EVENT,
              PathDelay       => tpd_PN_QN,
              Condition       => (CN_ipd = '1')),
    -- Third input pin that affects the output
      2   => (InputChangeTime => CN_ipd'LAST_EVENT,
              PathDelay       => tpd_CN_QN,
              Condition       => (PN_ipd = '0'))),
    GlitchData      => GlitchData_QN.
    DefaultDelay    => VitalZeroDelay01,-- Delay to be used if all path condition are FALSE
    Mode            => OnEvent,   -- Mode for glitch processing
    MsgOn           => TRUE,      -- Message control on glitch
    XOn             => TRUE,      -- X-generation on glitch
    MsgSeverity     => ERROR);
```

9.4.2 VitalPathDelay01Z

Procedure VitalPathDelay01Z schedules path delays on signals for which the transition to or from 'Z' is important (e.g., modeling of tri-state drivers). In addition to the basic capabilities provided by all path delay procedures,VitalPathDelay01Z performs result mapping of the output value (using the value specified by the actual associated with the OutputMap parameter) before scheduling this value on the signal. This result mapping is performed after transition-dependent delay selection but before scheduling the final output.

Example:

```
VitalPathDelay01Z(
  OutSignal       => Q,      -- Signal being scheduled
  OutSignalName => "Q",                 -- Name of the signal
  OutTemp         => Q_zd,   -- New signal value
  Paths           => (                  -- One path data for each input
                                        -- affecting the output
   -- First input pin that affects the output
   0  =>      (InputChangeTime => D_ipd'LAST_EVENT,
              PathDelay       => tpd_D_Q,
              Condition       => (Enable = '0')),
```

```
        -- Second input pin that affects the output
        1  =>      (InputChangeTime => Enable_ipd'LAST_EVENT,
                    PathDelay       => tpd_Enable_Q,
                    Condition       => (Enable = '1'))),
    GlitchData     => GlitchData_Q,
    MsgOn          => TRUE,
    XOn            => TRUE,
    Mode           => OnEvent,
    MsgSeverity    => ERROR,
    OutputMap      => "UX01WHLHX"); -- Pullup behavior
```

9.5 Delay selection in VITAL primitives

In addition to functional computation, the VITAL primitive procedures perform delay selection, glitch handling, and signal scheduling. The delay selection mechanism in the primitives is different from that used in the path delay procedures.

The delay selection algorithm used by the VITAL primitive procedures is based on the following selection criteria:

— If the new output value is dependent on multiple input values, the delay selected is the maximum of the delays from the dependent inputs.

— If the new output value is determined by either of the input values, the delay selected is the minimum of the delays from these inputs.

Delay selection in VITAL primitive procedures is accomplished by maintaining separate output times from each input signal and then selecting the appropriate output delay based on the preceding selection criteria. The new value is scheduled on the output using the selected delay.

Control of glitch handling is provided through a formal parameter.

Example:

Let
Ti0 be the time when the output will change based on a falling input
Ti1 be the time when the output will change based on a rising input

For an AND primitive,
An output going to a '1' value will be scheduled after the maximum of Ti1 times for each input
An output going to a '0' value will be scheduled after the minimum of Ti0 times for each input

However, for a NAND primitive,
An output going to a '1' value will be scheduled after the minimum of Ti0 times for each input
An output going to a '0' value will be scheduled after the maximum of Ti1 times for each input

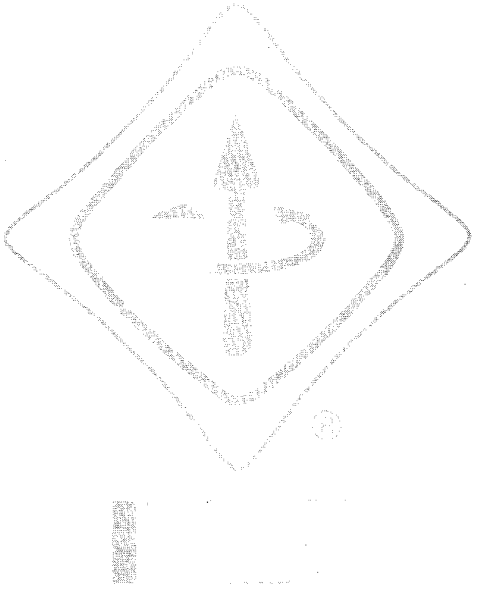
Similarly, for an OR primitive.
An output going to a '1' value will be scheduled after the minimum of Ti1 times for each input
An output going to a '0' value will be scheduled after the maximum of Ti0 times for each input

9.6 VitalExtendToFillDelay

The function VitalExtendToFillDelay is a utility that provides a set of six transition-dependent delay values, even though fewer delay values may have been explicitly provided.

Example:

```
CONSTANT tpd_Input_Output : VitalDelayType01;
-- This variable holds two delay values
VARIABLE tpd_Control_Output: VitalDelayType01Z;
-- This variable holds six delay values
    ...
tpd_Control_Output := VitalExtendToFillDelay(tpd_Input_Output);
```

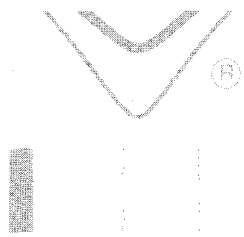


10. The VITAL standard packages

This standard defines two standard packages—VITAL_Timing and VITAL_Primitives—that predefine a number of items that are useful or required for designing VITAL-compliant models. These packages shall reside in the VHDL library IEEE.

The semantics of the VITAL standard packages are defined by their VHDL description according to IEEE Std 1076-1987 and IEEE Std 1164-1993. Their interfaces are defined by their package declarations, and their behavior is defined by the corresponding package bodies. An implementation may not add items, delete items, or otherwise alter the contents of the VITAL standard packages. An implementation may choose to implement the package bodies in a more efficient form; however, the resulting semantic shall not differ from the formal semantic provided herein.

The standard packages (subclauses 10.1 through 10.4) are on the diskette that is included with this standard. These standard packages are an official part of this standard. Please consult this diskette for the contents of these standard packages.



Annex A

(informative)

Syntax summary

VITAL_control_generic_declaration ::= [4.3.2.2]
[**constant**] identifier_list ::= [**in**] type_mark [index_constraint] [:= *static*_expression] ;

VITAL_design_file ::= [4.2.2]
VITAL_design_unit { VITAL_design_unit }

VITAL_design_unit ::= [4.2.2]
context_clause library_unit
| context_clause VITAL_library_unit

VITAL_entity_declarative_part ::= VITAL_Level0_attribute_specification [4.3]

VITAL_entity_generic_clause ::= [4.3]
generic (VITAL_entity_interface_list) ;

VITAL_entity_header ::= [4.3]
[VITAL_entity_generic_clause]
[VITAL_entity_port_clause]

VITAL_entity_interface_declaration ::= [4.3]
interface_constant_declaration
| VITAL_timing_generic_declaration
| VITAL_control_generic_declaration
| VITAL_entity_port_declaration

VITAL_entity_interface_list ::= [4.3]
VITAL_entity_interface_declaration { ; VITAL_entity_interface_declaration }

VITAL_entity_port_clause ::= [4.3]
port (VITAL_entity_interface_list) ;

VITAL_entity_port_declaration ::= [4.3.1]
[**signal**] identifier_list : [mode] type_mark [index_constraint] [:= *static*_expression] ;

VITAL_functionality_section ::= [6.4.3.2.2]
{ VITAL_variable_assignment_statement | procedure_call_statement }

VITAL_internal_signal_declaration ::= [6.3.1]
signal identifier_list : type_mark [index_constraint] [:= expression] ;

VITAL_Level_0_architecture_body ::= [4.4]

architecture identifier **of** *entity*_name **is**
VITAL_Level_0_ architecture_declarative_part
begin
architecture_statement_part
end [*architecture*_simple_name] ;

VITAL_Level_0_architecture_declarative_part ::= [4.4]
VITAL_Level0_attribute_specification { block_declarative_item }

VITAL_Level_0_entity_declaration ::= [4.3]
entity identifier **is**
VITAL_entity_header
VITAL_entity_declarative_part
end [*entity*_simple_name] ;

VITAL_Level_1_architecture_body ::= [6.2]
architecture identifier **of** *entity*_name **is**
VITAL_Level_1_architecture_declarative_part
begin
VITAL_Level_1_architecture_statement_part
end [*architecture*_simple_name] ;

VITAL_Level_1_architecture_declarative_part ::= [6.3]
VITAL_Level1_attribute_specification
{ VITAL_Level_1_block_declarative_item }

VITAL_Level_1_architecture_statement_part ::= [6.4]
VITAL_Level_1_concurrent_statement { VITAL_Level_1_concurrent_statement }

VITAL_Level_1_block_declarative_item ::= [6.3]
constant_declaration
| alias_declaration
| attribute_declaration
| attribute_specification
| VITAL_internal_signal_declaration

VITAL_Level_1_ concurrent_statement ::= [6.4]
VITAL_wire_delay_block_statement
| VITAL_negative_constraint_block_statement
| VITAL_process_statement
| VITAL_primitive_concurrent_procedure_call

VITAL_Level0_attribute_specification ::= attribute_specification [4.1]

VITAL_Level1_attribute_specification ::= attribute_specification

VITAL_library_unit ::= [4.2.2]
VITAL_Level_0_entity_declaration
| VITAL_Level_0_architecture_body
| VITAL_Level_1_architecture_body

VITAL_negative_constraint_block_statement ::= [6.4.2]
*block*_label :
block

```
        begin
            { VITAL_negative_constraint_concurrent_procedure_call }
        end block [ block_label ] ;

VITAL_primitive_concurrent_procedure_call ::=                                    [6.4.4]
        VITAL_primitive_concurrent_procedure_call

VITAL_process_declarative_item ::=                                               [6.4.3.1]
          constant_declaration
        | alias_declaration
        | attribute_declaration
        | attribute_specification
        | VITAL_variable_declaration

VITAL_process_declarative_part ::=                                               [6.4.3.2]
        { VITAL_process_declarative_item }

VITAL_process_statement ::=                                                      [6.4.3]
        [ process_label : ]
        process ( sensitivity_list )
            VITAL_process_declarative_part
        begin
            VITAL_process_statement_part
        end process [ process_label ] ;

VITAL_process_statement_part ::=                                                 [6.4.3.2]
        [ VITAL_timing_check_section ]
        [ VITAL_functionality_section ]
        [ VITAL_path_delay_section ]

VITAL_target ::= unrestricted_variable_name                                      [6.4.3.2.2]

VITAL_timing_check_condition ::= generic_simple_name                             [6.4.3.2.1]

VITAL_timing_check_section ::=                                                   [6.4.3.2.1]
        if VITAL_timing_check_condition then
            { VITAL_timing_check_statement }
        end if ;

VITAL_timing_check_statement ::= procedure_call_statement                        [6.4.3.2.1]

VITAL_timing_generic_declaration ::=                                             [4.3.2.1]
        [ constant ] identifier_list ::= [ in ] type_mark [ index_constraint ] [ := static_expression ] ;

VITAL_variable_assignment_statement ::=                                          [6.4.3.2.2 ]
        VITAL_target := expression ;

VITAL_variable_declaration ::=                                                   [6.4.3.1.1]
        variable identifier_list : type_mark [ index_constraint ] [ := expression ] ;

VITAL_wire_delay_block_statement ::=                                             [6.4.1]
        block_label :
        block
        begin
```

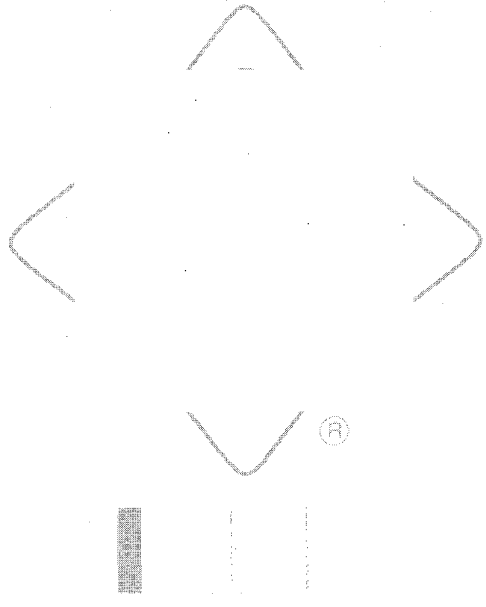
VITAL_wire_delay_block_statement_part
end block [*block*_label] ;

VITAL_wire_delay_block_statement_part ::= [6.4.1]
{ VITAL_wire_delay_concurrent_procedure_call
| VITAL_wire_delay_generate_statement }

VITAL_wire_delay_concurrent_procedure_call ::= concurrent_procedure_call [6.4.1]

VITAL_wire_delay_generate_parameter_specification ::= [6.4.1]
identifier **in** *range*_attribute_name

VITAL_wire_delay_generate_statement ::= [6.4.1]
*generate*_label :
for VITAL_wire_delay_generate_parameter_specification **generate**
{ VITAL_wire_delay_concurrent_procedure_call }
end generate [*generate*_label] ;



Annex B

(informative)

Glossary

This glossary contains brief, informal definitions of a number of hardware-specific terms and phrases that are used in this standard. The definitions in this annex are not a part of the formal definition of this standard.

ASIC cell: The building block of an Application-Specific Integrated Circuit (ASIC).

device delay: The intrinsic delay of a cell; it represents the delay associated from each input path to the given output of the cell.

hold time: The time period following a clock edge during which an input signal value may not change value.

interconnect path delay: Delays on the wires that connect various instantiations of ASIC cells in a design.

no change time: A stable interval associated with a setup or hold constraint. A signal checked against a control signal has to remain stable during the setup period established before the start of the control pulse, the entire width of the pulse, and the hold period established after the pulse. Each of these stable intervals is a no change time.

period: The time delay from the specified edge of a clock pulse to the corresponding edge of the following clock pulse.

propagation delay: The time delay from the arrival of an input signal value to the appearance of a corresponding output signal value.

pulse width: The time duration for which the value of signal remains unchanged at a low or high state.

recovery time: The minimal time interval by which a change to an unasserted value on an asynchronous (set, reset) input signal has to precede the clock edge.

removal time: The minimal time interval for which an asserted condition has to be present on an asynchronous (set, reset) input signal following the clock edge.

setup time: The time period prior to a clock edge during which an input signal value may not change value.

skew time: The maximum allowable delay between two signals. A delay that exceeds the skew time causes devices to behave unreliably.

Annex C

(informative)

Bibliography

[B1] IEEE Std 1076-1993, IEEE Standard VHDL Language Reference Manual (ANSI).[1]

[B2] IEEE Std 1076/INT-1991, IEEE Standards Interpretations: IEEE Std 1076-1987, IEEE Standard VHDL Language Reference Manual.

[B3] EIA-5670000-91, EIA Commercial Component Model Specification.[2]

[1]IEEE publications are available from the Institute of of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA.

[2]EIA publications are available from the Global Engineering Documents, 15 Inverness Way East, Englewood, CO 80112, USA.

Index