

IEEE Std 1076.4-2000

(Revision of
IEEE Std 1076.4-1995)

IEEE Standards

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

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee



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

28 September 2001

Print: SH94959
PDF: SS94959

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

Sponsor

**Design Automation Standards Committee
of the
IEEE Computer Society**

Approved 21 September 2000

IEEE-SA Standards Board

Abstract: The VITAL (VHDL Initiative Towards ASIC Libraries) ASIC Modeling Specification is defined in this standard. This modeling specification defines a methodology which 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.
3 Park Avenue, New York, NY 10016-5997, USA

Copyright © 2001 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 28 September 2001. Printed in the United States of America.

Print: ISBN 0-7381-2691-0 SH94959
PDF: ISBN 0-7381-2692-4 SS94959

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. The IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and serve without compensation. While the IEEE administers the process and establishes rules to promote fairness in the consensus development process, the IEEE does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

Use of an IEEE Standard is wholly voluntary. The IEEE disclaims liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other IEEE Standard document.

The IEEE does not warrant or represent the accuracy or content of the material contained herein, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or fitness for a specific purpose, or that the use of the material contained herein is free from patent infringement. IEEE Standards documents are supplied “**AS IS**.”

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.

In publishing and making this document available, the IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is the IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEEE Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

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 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 societies and Standards Coordinating 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 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. Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE-SA 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 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.

The IEEE and its designees are the sole entities that may authorize the use of IEEE-owned certification marks and/or trademarks to indicate compliance with the materials set forth herein.

Authorization to photocopy portions of any individual standard for internal or personal use is granted by the Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; (978) 750-8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Introduction

(This introduction is not part of IEEE Std 1076.4-2000, IEEE Standard for VITAL ASIC.)

The objectives of the VITAL (VHDL Initiative Towards ASIC Libraries) initiative can be summed up in one sentence:

Accelerate the development of sign-off quality ASIC macrocell simulation libraries written in VHDL by leveraging existing methodologies of model development.

The VITAL ASIC modeling specification is a revision of the IEEE 1076.4-1995, IEEE Standard for VITAL ASIC Modeling Specification. Several new modeling enhancements have been added to the standard and several usability issues which have been raised with the 1995 standard have been addressed. The new enhancements and usability improvements addressed include:

- Standardized ASIC memory models
- Support of IEEE VHDL93 and SDF 1497 standards
- Multisource interconnect timing simulation
- SKEW constraint timing checks
- Timing constraint checks feature enhancements
- Additional generics to control 'X' generation and message reporting for glitches and timing constraints.
- Negative constraint calculation enhancement for vector signals to support memory models
- Fast delay path disable
- Negative glitch preemption

These new features will improve the functional, timing accuracy significantly and aid performance of gate level VHDL simulations.

The major enhancement is the definition of a ASIC memory modeling standard. With the addition of memory model package VITAL_Memory, a standard is defined which allow memory models to be coded in VHDL more efficiently. The standard VITAL memory model package provides a method to represent memories, procedures and functions to perform various operations and the definition of a modeling style that promotes consistency, maintainability and tool optimization. This standard does not define modeling behavior of specific memories. The scope of the memory model standard is currently restricted to ASIC memory modeling requirement for static RAMs and ROMs. The VITAL memory modeling enhancements are specified in Clause 10 through Clause 12. The VITAL standard memory package is found in Clause 13.

The memory model standard is derived from contributed work from the LSI Logic VHDL behavioral model and Mentor Graphics Memory Table Model (MTM) techniques. The generous support of VHDL International provided the needed funding to take these two contributed works and convert them into the memory specification and package code by the IEEE 1076.4 TAG (Technical Action Group) with significant contribution coming from leading EDA services company GDA Technologies.

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 specification are the responsibility of the IEEE 1076.4 TAG. This group consists of Ekambaram Balaji, Prakash Bare, Nitin Chowdhary, Jose De Castro, Martin Gregory, Rama Kowsalya, B. Sudheendra Phani Kumar, William Yam, David Lin, Ashwini Mulgaonkar, Ajayharsh P. Varikat, and Steve Wadsworth and is chaired by Dennis B. Brophy. Without the dedication and hard work of this group it would not have been possible to complete this work.

The VITAL effort germinated from ideas generated at the VHDL International Users' Forum held in Scottsdale, Arizona in May 1992. Further discussions brought people to the conclusion 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 is viewed as a catalyst for the rapid deployment of ASIC libraries once this initiative is standardized under the IEEE.

The 1076.4 Working Group has a large membership of over three hundred 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 specification. This group is chaired by Victor Berman.

The VITAL ASIC modeling specification is the result of numerous discussions with ASIC vendors, EDA 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 except for multi-source interconnect.
- To provide a set of modeling rules (Level 1) which constrain the use of VHDL to a point that is amenable for simulator optimizations, and at the same time 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 model's generic parameter list or via SDF direct import.

Participants

The following persons were members of the 1076.4 Technical Action Group (TAG):

Victor Berman, Chair
Dennis B. Brophy, Chair, 1076.4 Technical Action Group

Ekambaram Balaji
Prakash Bare
Nitin Chowdhary
Jose De Castro

Martin Gregory
Rama Kowsalya
Phani Kumar
David Lin,
Ashwini Mulgaonkar

B. Sudheendra
Ajayharsh P. Varikat
Steven D. Wadsworth
William Yam

The following members of the balloting committee voted on this standard:

Peter J. Ashenden
Stephen A. Bailey
David L. Barton
Victor Berman
J Bhasker
William Billowitch
Dennis B. Brophy
Brian A. Dalio
Timothy R. Davis
Ted Elkind
Andrew Guyler

Rich Hatcher
Jim Heaton
Neil G. Jacobson
Osamu Karatsu
Jake Karrfalt
Satoshi Kojima
Gunther Lehmann
Maqsoodul Mannan
Paul J. Menchini
Jean P. Mermet

Quentin G. Schmierer
Steven E. Schulz
Francesco Sforza
Joseph P. Skudlarek
Joseph J. Stanco
Steven D. Wadsworth
Ronald Waxman
Ron Werner
John M. Williams
John Willis
Mark Zwolinski

When the IEEE-SA Standards Board approved this standard on 21 September 2000, it had the following membership:

Donald N. Heirman, *Chair*
James T. Carlo, *Vice Chair*
Judith Gorman, *Secretary*

Satish K. Aggarwal
Mark D. Bowman
Gary R. Engmann
Harold E. Epstein
H. Landis Floyd
Jay Forster*
Howard M. Frazier
Ruben D. Garzon

James H. Gurney
Richard J. Holleman
Lowell G. Johnson
Robert J. Kennelly
Joseph L. Koepfinger*
Peter H. Lips
L. Bruce McClung
Daleep C. Mohla

James W. Moore
Robert F. Munzner
Ronald C. Petersen
Gerald H. Peterson
John B. Posey
Gary S. Robinson
Akio Tojo
Donald W. Zipse

*Member Emeritus

Also included is the following nonvoting IEEE-SA Standards Board liaison:

Alan Cookson, *NIST Representative*
Donald R. Volzka, *TAB Representative*

Andrew D. Ickowicz
IEEE Standards Project Editor

Contents

1.	Overview	1
1.1	Scope	1
1.2	Purpose	1
1.3	Intent of this standard	1
1.4	Structure and terminology of this standard	1
1.5	Syntactic description	2
1.6	Semantic description	3
1.7	Front matter, examples, figures, notes, and annexes	3
2.	References	3
3.	Basic elements of the VITAL ASIC modeling specification	4
3.1	VITAL modeling levels and compliance	4
3.2	VITAL standard packages	5
3.3	VITAL specification for timing data insertion	5
4.	The Level 0 specification	7
4.1	The VITAL_Level0 attribute	7
4.2	General usage rules	7
4.3	The Level 0 entity interface	8
4.4	The Level 0 architecture body	17
5.	Backannotation	19
5.1	Backannotation methods	19
5.2	The VITAL SDF map	20
6.	The Level 1 specification	35
6.1	The VITAL_Level1 attribute	35
6.2	The Level 1 architecture body	35
6.3	The Level 1 architecture declarative part	36
6.4	The Level 1 architecture statement part	36
7.	Predefined primitives and tables	46
7.1	VITAL logic primitives	46
7.2	VitalResolve	48
7.3	VITAL table primitives	48
8.	Timing constraints	54
8.1	Timing check procedures	54
8.2	Modeling negative timing constraints	59
9.	Delay selection	70
9.1	VITAL delay types and subtypes	70

9.2	Transition dependent delay selection	71
9.3	Glitch handling.....	71
9.4	Path delay procedures	72
9.5	Delay selection in VITAL primitives	74
9.6	VitalExtendToFillDelay.....	75
10.	The Level 1 Memory specification	76
10.1	The VITAL Level 1 Memory attribute	76
10.2	The VITAL Level 1 Memory architecture body.....	76
10.3	The VITAL Level 1 Memory architecture declarative part.....	77
10.4	The VITAL Level 1 Memory architecture statement part.....	77
11.	VITAL Memory function specification	87
11.1	VITAL memory construction	87
11.2	VITAL memory table specification	90
11.3	VitalDeclareMemory	99
11.4	VitalMemoryTable.....	101
11.5	VitalMemoryCrossPorts	103
11.6	VitalMemoryViolation.....	105
12.	VITAL memory timing specification	108
12.1	VITAL memory timing types	108
12.2	Memory Output Retain timing behavior.....	109
12.3	VITAL Memory output retain timing specification.....	110
12.4	Transition dependent delay selection.....	110
12.5	VITAL memory path delay procedures	111
12.6	VITAL memory timing check procedures.....	116
13.	The VITAL standard packages	121
13.1	VITAL_Timing package declaration	121
13.2	VITAL_Timing package body	136
13.3	VITAL_Primitives package declaration	163
13.4	VITAL_Primitives package body	232
13.5	VITAL_Memory package declaration	302
13.6	VITAL_Memory package body	323
Annex A	(informative) Syntax summary	412
Annex B	(informative) Glossary	418
Annex C	(informative) Bibliography	420

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

1. Overview

This clause describes the purpose and organization of this standard.

1.1 Scope

To provide a standard method of modeling ASICs in VHDL. This method is aimed at providing efficient, accurate, and tool independent simulation suitable for large chip-level designs typical of those which are based on ASICs.

1.2 Purpose

Current industry methods for designing complex chip-level designs rely on proprietary solutions which are based on specific commercial tools. This standard provides an effective means of performing those designs in a standard, non-proprietary manner that is independent of specific tools. This promotes cost effective design flows and promotes healthy levels of competition in the electronic design industry. This standard builds on the work of IEEE 1076 VHDL which is a standard hardware description language designed to allow such tool independent electronic design.

1.3 Intent of this standard

The intent of this standard is to accurately define the Draft Standard VITAL ASIC Modeling Specification. The primary audiences of this standard are the implementors of tools supporting the specification and ASIC modelers.

1.4 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.

1.5 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 (BNF); in particular:

- a) Lower cased 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 the IEEE Std 1076-1993 VHDL LRM.
- j) A syntactic category beginning with the unitalicized prefix “VITAL_” represents a subset of a VHDL syntactic category.

1.6 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 non-compliant 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 non-compliant 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 non-compliant modeling practice; implementations are required to detect the condition and report an error to the user of the tool.

1.7 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 manual but are not part of the definition of the Draft Standard VITAL ASIC Modeling Specification.

Some subclauses of this definition 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 clause lists the standards upon which this standard depends. Bibliographic references may be found in Annex C. Citations of the form “[C1]” refer to items listed in Annex C, not to items listed in this clause.

IEEE Std 1076-1993, IEEE Standard VHDL Language Reference Manual.¹

IEEE Std 1164-1993, IEEE Standard Multivalued Logic System for VHDL Model Interoperability (Std_logic_1164).

IEEE P1497, Draft Standard Delay Format Specification.

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

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.

The VITAL ASIC modeling specification is an application of the VHSIC Hardware Description Language (VHDL), described in IEEE Std 1076-1993 [C1]². This document uses the term *VHDL* to refer to the VHSIC Hardware Description Language.

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

This modeling specification relies on the IEEE P1497 Draft Standard Delay Format (SDF), as a standard external timing data representation. Throughout this document, the term *SDF* refers to this particular version of the delay format.

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

NOTE—VHDL is used to define the VITAL ASIC modeling specification except certain features of multi-source interconnect modeling, direct SDF backannotation and negative constraint calculations. The use of the VITAL package code without specific implementation of these other features does not offer full use of these features.

3.1 VITAL modeling levels and compliance

A VITAL ASIC cell is represented by a VHDL design entity. The VITAL ASIC modeling specification 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.

The specification defines three modeling levels; these levels are called *VITAL Level 0*, *VITAL Level 1* and *VITAL Level 1 Memory*. 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 and VITAL Level 1 Memory specification. The modeling rules for a VITAL Level 1 specification and VITAL Level 1 Memory specification are mutually exclusive.

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 which adheres to the rules in the Level 0 specification is said to be a *VITAL Level 0 model*. The Level 0 modeling specification is described in Clause 4.

²The numbers in brackets correspond to those of the bibliography in Annex C.

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 which adheres to the rules in both the Level 0 model interface specification and the Level 1 model architecture specification is said to be a *VITAL Level 1 model*. The Level 1 modeling rules are defined in Clause 6.

The VITAL Level 1 Memory specification defines a usage model for constructing complete memory cell models in a manner that facilitates optimization of the execution of the memory models. A model which adheres to the rules in both the Level 0 model interface specification and the Level 1 Memory model architecture specification is said to be a *VITAL Level 1 Memory model*. The Level 1 Memory modeling rules are defined in clause 10.

A model that is a VITAL Level 0 model or a VITAL Level 1 model or a VITAL Level 1 Memory 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, 6.1 and 10.1.

NOTES:

- 1) A Level 1 model or a Level 1 Memory model is by definition a Level 0 model as well (but not vice versa).
- 2) The rules outlined in the Level 0, Level 1 and Level 1 Memory specifications apply to model descriptions, not to the VITAL standard packages themselves.
- 3) AVITAL 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 correctly execute a VITAL compliant model, and to identify and reject models which are not compliant. A VITAL compliant tool is also expected to fully support the processes described in the specification, including SDF backannotation and negative time sequential constraint transformation.

3.2 VITAL standard packages

The Draft Standard VITAL ASIC Modeling Specification defines three standard packages for use in specifying the timing and functionality of a model: *VITAL_Timing*, *VITAL_Primitives* and *VITAL_Memory*. The text of these packages may be found in Clause 13.

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.

The *VITAL_Memory* package defines data types and subprograms to support development of memory models. Included in this package are routines for functional modeling, corruption handling, memory specific delay selection, output scheduling, and timing violation checking and reporting.

3.3 VITAL specification for timing data insertion

The Draft Standard VITAL ASIC Modeling Specification defines certain semantics that are assumed by a VITAL compliant model and must be implemented by a tool processing or simulating VITAL compliant models which 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.

The specification 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 non-negative ones for the purpose of simulation.

4. The Level 0 specification

The 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 Level 0 design entity descriptions, and standardizing the specification and processing of timing information. General 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 Level 0 entity or architecture is identified by its decoration with the VITAL_Level0 attribute, which indicates an intention to adhere to the Level 0 specification.

VITAL_Level0_attribute_specification ::= attribute_specification

A Level 0 entity or architecture shall contain a specification of the VITAL_Level0 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 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 Level 1 architecture or Level 1 Memory architecture, which is also by definition a Level 0 architecture, contains a VITAL_Level1 or VITAL_Level1_Memory attribute specification (see 6.1 and 10.1) rather than a VITAL_Level0 attribute specification. The above rules apply to architectures that are only Level 0.

Example:

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

4.2 General usage rules

A 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 shall use IEEE Std 1076-1993 features. Use of foreign architecture bodies or package bodies is prohibited.

It is erroneous for a VITAL model to make use of vendor-supplied attributes or other non-normative 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. The VITAL ASIC modeling specification imposes no special requirement on the placement of VITAL compliant description within design files, which may contain a mixture of compliant and non-compliant 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
  | VITAL_Level_1_memory_architecture_body
```

4.3 The 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] [ 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 statement is 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 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 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

The VITAL ASIC modeling specification 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 a model's timing generics and 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.

The Draft Standard VITAL ASIC Modeling Specification 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 which 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 which 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 which 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, 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 a scalar, the type of the timing generic shall be a scalar form of delay type. If the timing generic is associated with two scalar ports, 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, 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 a vector, 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 a vector, the type of the timing generic shall be a vector form of delay type. If the number of scalar subelements in the first port is not equal to the number of scalar subelements in the second port, 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. If the number of scalar subelements in the first port is equal to the number of scalar subelements in the second port, the length of the index range of the generic shall be equal to one of the following:
 - (a) The product of the number of scalar subelements in the first port and the number of scalar subelements in the second port.
 - (b) The number of scalar subelements in the first port.
- If the timing generic is a scalar form of delay type and it is associated with a single port then that port shall be a scalar port. If the timing generic is a scalar form of delay type and it is associated with two ports then both those ports shall be scalar ports.

NOTE—These consistency requirements between timing generics and ports 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. This generic may include an output retain timing specification. Its name is of the form

```
<VITALPropagationDelayName> ::=
    tpd_<InputPort>_<OutputPort> [ <SDFSSimpleConditionAndOrEdge> ]
```

The type of a propagation delay generic shall be a VITAL delay type (see 9.1). In the presence of output retain delays the propagation delay generic type shall only be **VitalDelayType01Z** (**VitalDelayArrayType01Z**) or **VitalDelayType01ZX** (**VitalDelayArrayType01ZX**).

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> [ <SDFFFullConditionAndOrEdge> ]
```

The type of a 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> [ <SDFFFullConditionAndOrEdge> ]
```

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> [ <SDFFFullConditionAndOrEdge> ]
```

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> [_<SDFSsimpleConditionAndOrEdge>]

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> [_<SDFSsimpleConditionAndOrEdge>]

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 a 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 **tnsetup** is a backannotation timing generic representing the setup time associated with a no change timing constraint. Its name is of the form

<VITALNoChangeSetupTimeName> ::=
tnsetup_<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 which 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 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> [ _<SDFSsimpleConditionAndOrEdge> ]
```

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. The size of the biased propagation delay generic shall be equal to the product of the size of the corresponding propagation delay generic and the size of the clock port.

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 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

The VITAL ASIC modeling specification 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** generic 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 generics **XOn**, **XOnGlitch**, and **MsgOn**, **MsgOnGlitch** are switches that may be used as standard mechanism for control of 'X' generation and assertion messages relating to glitch violations.

A generic with the name **XOn** or **XOnGlitch** shall be of the predefined type **Boolean**. It may be used to control 'X' generation for path delays. The value **TRUE** indicates that glitch violations should cause certain output ports to be assigned value 'X'. The value **FALSE** means that 'X' should not be generated on the output ports for associated glitch violations.

A generic with the name **MsgOn** or **MsgOnGlitch** shall be of the predefined type **Boolean**. It may be used to control assertion messages for path delays. The value **TRUE** indicates that assertion messages should be issued when glitch violations are encountered. The value **FALSE** means assertion messages should not be issued.

The generics **XOn**, **XOnChecks**, and **MsgOn**, **MsgOnChecks** are switches that may be used as standard mechanism for control of 'X' generation and assertion messages relating to timing violations.

A generic with the name **XOn** or **XOnChecks** shall be of the predefined type **Boolean**. It may be used to control 'X' generation for timing checks. The value **TRUE** indicates that timing violations should cause certain output ports to be assigned value 'X'. The value **FALSE** means that 'X' should not be generated on the output ports for associated timing violations. The value of the control generic by itself only affects the formal parameter 'Violation' of all timing check procedures in the VITAL standard timing package.

A generic with the name **MsgOn** or **MsgOnChecks** shall be of the predefined type **Boolean**. It may be used to control assertion messages for timing checks. The value **TRUE** indicates that assertion messages should be issued when timing violations are encountered. The value **FALSE** means assertion messages should not be issued.

:NOTES:

1) The declaration of a control generic by itself has no effect; the generic must 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 567A [C3] XGeneration and MGeneration features. In particular, declaration of an **XOn** or **MsgOn** generic does NOT automatically enable timing checks.

4.4 The 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] [ architecture_simple_name ] ;

VITAL_Level_0_architecture_declarative_part ::=
  VITAL_Level0_attribute_specification { block_declarative_item }
```

The entity associated with a 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` and `VITAL_Memory` 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 which 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 which adhere to the 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. VHDL configuration declarations cannot be used to annotate multi-source interconnect path delays. 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 which 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 directly modify the backannotation timing generics in a VITAL compliant model. 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 must be appropriate for the type class of the corresponding VHDL timing generic, and all applicable VHDL constraints on the value or set of values must 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 which 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 of the VITAL ASIC modeling specification related to the SDF mapping or backannotation process. Its behavior after reporting an error is implementation defined.

Certain SDF constructs are not supported by the VITAL ASIC modeling specification; 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 which was set during standard VHDL elaboration shall remain in effect.

NOTE—AVITAL 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 which 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 which 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. 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 correctly read and interpret the SDF file. 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 correctly interpret the SDF file.

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 version 4.0.

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* {*timing_spec*})

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

- 1) The *cell_instance* and *celltype* constructs are used to locate a path or a set of paths in the VHDL design hierarchy which correspond to the instance(s) to which the data applies.
- 2) 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 the Draft Standard VITAL ASIC Modeling Specification.

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) which 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 which 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** [Hierarchical Identifier])
 | (**INSTANCE** *)

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 which identifies all component instances, in or below the current level of the design hierarchy, which 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**. **c1** must 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 *delval_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 at a time 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 which 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 *delval_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 *delval_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 Table [C1].

In the presence of RETAIN delay specification the mapping of each element of the array to the corresponding SDF value is described in Table [C2]. Following restriction are placed on the usage of RETAIN delays:

- In the presence of RETAIN delay specification, the number of transition dependant delay values shall be limited to 6 transitions.
- If RETAIN delay specification includes three different rvalues (r1,r2,r3) the corresponding generic type must be **VitalDelayType01ZX**.
- If RETAIN delays are specified alongwith tristate IOPATH delays, the corresponding generic type must be **VitalDelayType01ZX**.

In the following table, each row of the table represents a form of SDF *delval_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

VITAL transition dependent delay value												
SDF <i>delval_list</i>	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 Annotator follows the SDF annotation rules regarding null delay values and extension of lists of rvalues.

Table 2— Mapping of SDF retain delay values to VITAL transition dependent delay values

VITAL transition dependent delay value												
SDF <i>retval_list</i>	tr01	tr10	tr0z	trz1	tr1z	trz0	tr0x	trx1	tr1x	trx0	trxz	trzx
r1 v1	v1	v1	r1	v1	r1	v1						
r1 r2 v1	v1	v1	r1	v1	r2	v1						
r1 r2 r3 v1	v1	v1	v1	v1	v1	v1	r1		r2			r3
r1 v1 v2	v1	v2	r1	v1	r1	v2						
r1 r2 v1 v2	v1	v2	r1	v1	r2	v2						
r1 r2 r3 v1 v2	v1	v2	v1	v1	v2	v2	r1		r2			r3
r1 v1 v2 v3	v1	v2	v3	v1	v3	v2	r1		r1			r1
r1 r2 v1 v2 v3	v1	v2	v3	v1	v3	v2	r1		r2			min (r1, r2)
r1 r2 r3 v1 v2 v3	v1	v2	v3	v1	v3	v2	r1		r2			r3
r1 v1 v2 v3 v4 v5 v6	v1	v2	v3	v4	v5	v6	r1		r1			r1
r1 r2 v1 v2 v3 v4 v5 v6	v1	v2	v3	v4	v5	v6	r1		r2			min (r1, r2)
r1 r2 r3 v1 v2 v3 v4 v5 v6	v1	v2	v3	v4	v5	v6	r1		r2			r3
v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10	v11	v12

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 the Draft Standard VITAL ASIC Modeling Specification, 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* {*del_def*})
||= (**INCREMENT** *del_def* {*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_instance {retain_def} delval_list )
          ||= ( COND conditional_port_expr ( IOPATH port_spec port_instance {retain_def}
delval_list ) )
retain_def ::= ( RETAIN retval_list )

delval ::=
    rvalue
    | ( rvalue rvalue )
    | ( rvalue rvalue rvalue )
delval_list ::=
    delval
    | delval delval
    | delval delval delval
    | delval delval delval delval [ delval ] [ delval ]
    | delval delval delval delval delval delval delval [ delval ] [ delval ] [ delval ] [ delval ] [ delval ]
retval_list ::=
    delval
    | delval delval
    | delval delval delval

```

NOTE—Although the SDF specification allows for RETAIN times to be a *delval_list* or *retval_list*, the VITAL Level0 backannotation rules does not allow the usage of *delval_list* for RETAIN times.

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

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

The <InputPort> corresponds to the port name specified in the *port_spec*, and the <OutputPort> corresponds to the port specified in the *port_path*. The <SDFSsimpleConditionAndOrEdge> 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

$$del_def ::= (\textbf{PORT} \textit{port_instance} \textit{delval_list})$$

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*.

If the SDF backannotator detects multiple INTERCONNECT delay entries which map to this generic hence indicating multiple sources of this input port, the backannotation of the PORT entry will proceed according to section 5.2.7.1.5.

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

$$del_def ::= (\textbf{INTERCONNECT} \textit{port_instance} \textit{port_instance} \textit{delval_list})$$

$$\begin{aligned} \textit{port_instance} & ::= \textit{port_path} \\ & ::= \textit{instance port_path} \end{aligned}$$

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:

- In the default mode, the SDF Annotator will mark this instance of the interconnect path delay generic as being in multi source mode and store the delay(s) corresponding to this source of the input port in an internal list. The tipd_<Port> generic corresponding to this input port need not be updated in this case. This behavior is described in section 5.2.7.1.5.
- The system can optionally support a mode in which the SDF Annotator shall provide a control 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 Backannotation in multiple source mode

In the multi source mode of operation, during elaboration, an internal array of the following form is created for each instance of the call to VitalWireDelay (in a Level-1 compliant architecture) the port connected to which has multiple sources:

internal array : array(0 to (Number of sources of <Port> - 1)) of <Type>;

Where the interconnect delay from each source is maintained in one location of this array, and where <Type> is one of VitalDelayArrayType, VitalDelayArrayType01 or VitalDelayArrayType01Z depending upon the declaration of the corresponding interconnect path delay generic (tipd_<Port>).

During backannotation, the following rules are applied:

- a) On detecting multiple INTERCONNECT entries mapping to the same instance of a interconnect path delay generic, enable multi source mode for this instance of the generic.
- b) Create the internal array of delays as shown above with the currently backannotated value of the interconnect path delay generic annotated to each location in this array.
- c) Each INTERCONNECT entry (including the current one) updates the location in the internal array corresponding to the source indicated in this entry.
- d) Each PORT entry updates all the locations in the internal array.

5.2.7.1.6 DEVICE delays

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. The VITAL ASIC modeling specification supports only those device entries which 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*] *delval_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 )
         ||= ( REMOVAL port_tchk port_tchk rvalue )
         ||= ( RECREM port_tchk port_tchk rvalue rvalue )
         ||= ( SKEW port_tchk port_tchk rvalue )
         ||= ( BIDIRECTSKEW port_tchk port_tchk rvalue 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 the VITAL ASIC modeling specification.

Each true timing check definition (i.e. each *tc_def* which 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_tchks*) 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 following manner:

- 1) 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)

REMOVAL	tremoval	(see 4.3.2.1.3.5)
RECREM	trecovery	
	tremoval	
SKEW	tskew	(see 4.3.2.1.3.8)
BIDIRECTSKEW	tskew	
	tskew	
WIDTH	tpw	(see 4.3.2.1.3.7)
PERIOD	tperiod	(see 4.3.2.1.3.6)
NOCHANGE	tnctestup	(see 4.3.2.1.3.9)
	tnchold	(see 4.3.2.1.3.10)

For BIDIRECTSKEW between two ports, both the corresponding skew generics between the ports shall be updated.

- 2) 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.
- 3) 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.

$$\begin{aligned} port_tchk & ::= port_spec \\ & ||= (\text{COND} [qstring] timing_check_condition port_spec) \end{aligned}$$

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

$$\langle \text{ConditionNameEdge} \rangle [_ \langle \text{SDFSimpleConditionAndOrEdge} \rangle]$$

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

$$[\langle \text{ConditionName} \rangle _] \text{noedge}$$

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 which 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* *delval_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 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:

- 1) Separate each part of the condition with an underscore, removing any white space

2) 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

3) 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

4) 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_instance ::= 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 which 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 which has a single port specification that has an index or range specification. This scheme also applies to an SDF entry which 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:

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

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

NOTES:

1) For a generic with a descending range constraint of ($g1$ downto $g2$), the index computation reduces to

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

2) For a generic with an ascending range constraint of ($g1$ to $g2$), the index computation reduces to

$$x = g2 - \text{abs}(c - j2)$$

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:

If the generic is a timing generic, where the number of scalar subelements in the first and second ports are equal, and the generic size is equal to the number of scalar subelements in the first port, the index is computed as follows :

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

It is an error if $\text{abs}(r - i2) \neq \text{abs}(c - j2)$.

Otherwise, the index is computed as follows:

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

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

NOTES:

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

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

or

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

2) For a generic with an ascending range constraint of (g1 to g2), the index computation reduces to

$$x = g2 - \text{abs}(c - j2)$$

or

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

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).

Assuming the following VHDL declarations

```
generic (tpd_DATAIN_DATAOUT : VitalDelayArrayType01 ( 0 to 3 ) := (others => (0 ns, 0 ns)));
port ( DATAIN : IN std_logic_vector( 0 to 3 );
      DATAOUT : OUT std_logic_vector( 1 to 4 ) );
```

the SDF entry

```
(IOPATH DATAIN[0] DATAOUT[1] (10) (20))
```

will cause the SDF Annotator to annotate the delay value (10, 20) onto the generic subelement tpd_DATAIN_DATAOUT(0).

6. The Level 1 specification

The Level 1 specification is a set of modeling rules which 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 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 Level 1 architecture is identified by its decoration with the VITAL_Level1 attribute, which indicates an intention to adhere to the Level 1 specification.

VITAL_Level1_attribute_specification ::= attribute_specification

A Level 1 architecture shall contain a specification of the VITAL_Level1 attribute corresponding to the declaration of that attribute in package VITAL_Timing. 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 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 Level 0 specification, except for the declaration of the VITAL_Level0 attribute.

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

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

The use of subprogram calls and operators in a 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

References to deferred constants are not allowed.

6.3 The Level 1 architecture declarative part

The Level 1 architecture declarative part contains declarations of items that are available for use within the 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 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 Level 1 architecture shall contain at most one wire delay block statement.

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

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

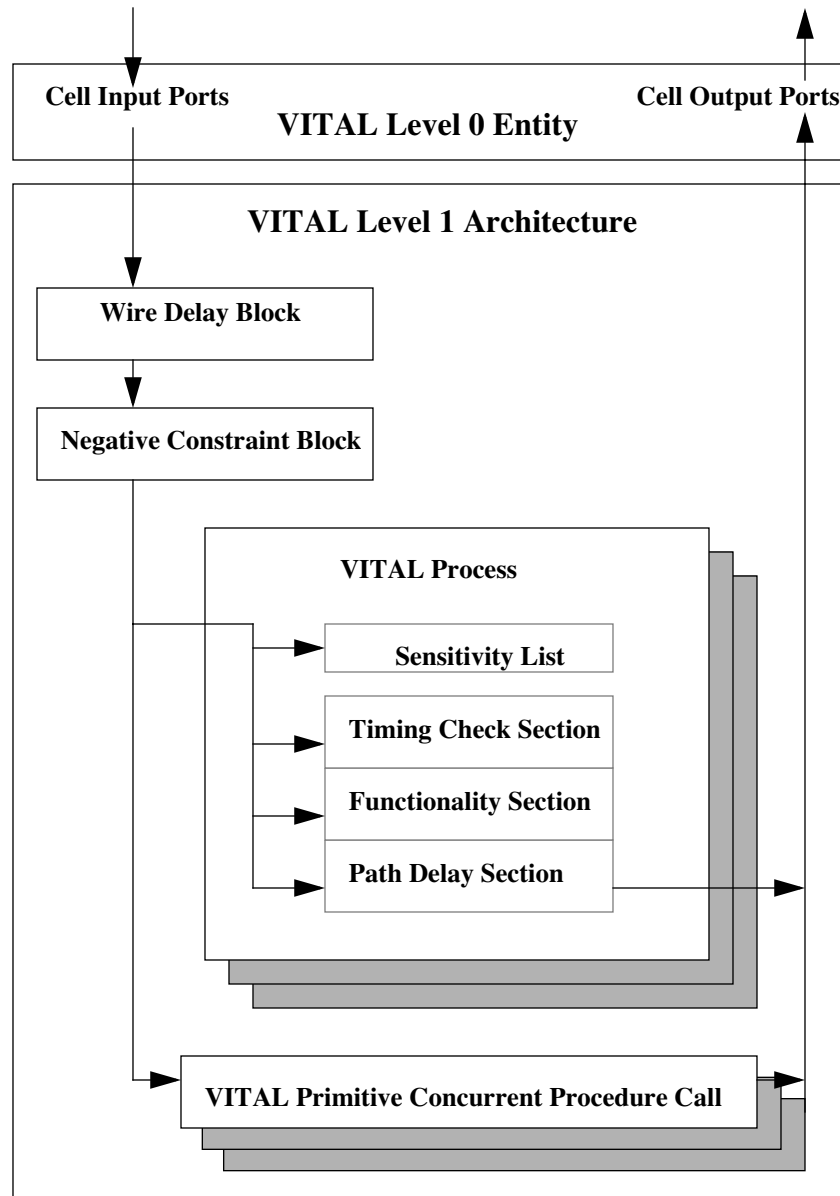


Figure 1—Structure of a VITAL Level 1 model

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 which 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 non-negative.

If the interconnect path delay generic associated to an instance of a **VitalWireDelay** call is in the multiple source mode (see 5.2.7.1.5 for details of multiple source mode), the simulation of this call to **VitalWireDelay** changes as follows:

- On every event on the actual associated to the **InSig** parameter, select the set of sources of this actual which have changed in this delta cycle.
- For each source load the location (see 5.2.7.1.5 for the details of the location created) created during SDF backannotation. This location can contain 1 (**VitalDelayType**), 2 (**VitalDelayType01**) or 6 (**VitalDelayType01Z**) delays. Apply the delay selection rules outlined in the corresponding **VitalCalcDelay** function declared in package **VITAL_Timing** using the new and the old value of the actual of the **InSig** parameter.

NOTE—This is equivalent to the call **<Source Delay> := VitalCalcDelay(<Source Location>, <New Value>, <Old Value>);** where the type of **<Source Location>** has been set depending upon the number of delays in that location.

- Select the minimum of all the source delays.
- The new value of the actual associated to the **InSig** parameter is assigned to the parameter associated to the **OutSig** parameter with this selected delay using transport mode.

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_block_statement_part
    end block [ block_label ] ;

VITAL_negative_constraint_block_statement_part ::=
    { VITAL_negative_constraint_concurrent_procedure_call
    | VITAL_negative_constraint_generate_statement_part }

VITAL_negative_constraint_generate_statement_part ::=
    VITAL_negative_constraint_generate_statement {
    VITAL_negative_constraint_generate_statement }

VITAL_negative_constraint_generate_statement ::=
    generate_label :
    for VITAL_negative_constraint_generate_parameter_specification generate
        { VITAL_negative_constraint_concurrent_procedure_call }
    end generate { generate_label }

VITAL_negative_constraint_generate_parameter_specification ::=
    identifier in range_attribute_name

VITAL_negative_constraint_concurrent_procedure_call ::= concurrent_procedure_call

```

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 internal clock delay or internal signal delay for a scalar port is modeled with a **VitalSignalDelay** procedure call.

Internal clock delay for a vector port shall be 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, the prefix of the attribute shall denote the clock port associated with the internal clock delay. The only statement within the generate statement shall be a **VitalSignalDelay** 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.

Internal signal delay for a vector port shall be modeled with a generate statement of a specific form. The generate statement shall be either a simple generate statement or a nested generate statement.

The simple generate statement shall have a generate parameter specification in which the discrete range is a predefined **Range** attribute, the prefix of the attribute shall denote the input port associated with the internal signal delay. The only statement within the generate statement shall be a **VitalSignalDelay** 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 nested generate statement, shall have a generate parameter specification in which the discrete range is a predefined **Range** attribute. The prefix of the attribute shall denote either the input port or the clock port. The only statements within the generate parameter specification shall be a simple generate statement.

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 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 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
SkewData	on procedures VitalInPhaseSkewCheck, VitalOutPhaseSkewCheck

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