IEEE Standard VHDL Language Reference Manual

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

1076™

IEEE
3 Park Avenue
New York, NY 10016-5997, USA

26 January 2009

IEEE Std 1076™-2008
(Revision of
IEEE Std 1076-2002)

IEEE Std 1076™-2008
(Revision of
IEEE Std 1076-2002)

IEEE Standard VHDL Language Reference Manual

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society

Approved 26 September 2008
IEEE SA-Standards Board

Acknowledgments

The editing and technical work done on the 2008 revision of this standard was done in collaboration between Accellera VHDL Technical Subcommittee and the IEEE VHDL Analysis and Screening Committee (VASG). Accellera had donated all of its work and copyrights back to the IEEE.

Material in clause titled “Protect Tool Directives” is derived from the document titled “A Mechanish for VHDL Source Protection” © 2004, Cadence Design Systems Inc. Used, modified, and reprinted by permission of Cadence Design Systems Inc.

The packages FIXED_GENERIC PKG, FIXED_PKG, FLOAT_GENERIC_PKG, FLOAT_PKG, and FIXED_FLOAT_TYPES were modified and used with permission from Eastman Kodak Company © 2006. Material in annex clauses titled “Using the fixed-point package” and “Using the floating-point package” is derived from the documents titled “Fixed Point Package User’s Guide” and “Floating Point Package User’s Guide” by Eastman Kodak Company © 2006. Used, modified, and reprinted by permission of Eastman Kodak Company.

The package STD_LOGIC_TEXTIO was modified and used with permission of Synopsys, Inc. © 1990, 1991, and 1992.

Abstract: VHSIC Hardware Description Language (VHDL) is defined. VHDL is a formal notation intended for use in all phases of the creation of electronic systems. Because it is both machine readable and human readable, it supports the development, verification, synthesis, and testing of hardware designs; the communication of hardware design data; and the maintenance, modification, and procurement of hardware. Its primary audiences are the implementors of tools supporting the language and the advanced users of the language.

Keywords: computer languages, electronic systems, hardware, hardware design, VHDL

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

PDF: ISBN 978-0-7381-5800-6 STD95821
CD-ROM: ISBN 978-0-7381-5801-3 STDCD95821

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 or the soundness of any judgments 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 his or her independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

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. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position, explanation, or interpretation of the IEEE.

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
Piscataway, NJ 08854
USA

Introduction

This introduction is not part of IEEE Std 1076-2008, IEEE Standard VHDL Language Reference Manual.

The VHSIC Hardware Description Language (VHDL) is a formal notation intended for use in all phases of the creation of electronic systems. Because it is both machine readable and human readable, it supports the development, verification, synthesis, and testing of hardware designs; the communication of hardware design data; and the maintenance, modification, and procurement of hardware.

This document, IEEE Std 1076-2008, is a revision of IEEE Std 1076-2002 as amended by IEEE Std 1076c™-2007. Initial work on gathering requirements and developing language extensions was undertaken by the IEEE VHDL Analysis and Standardization Group (VASG), otherwise known as the 1076 Working Group. Subsequently, Accellera[a] sponsored an effort to complete that work and draft a revised Language Reference Manual. That draft was returned to IEEE for final revision and approval, resulting in this document and the associated machine-readable files. This revision incorporates numerous enhancements, both major and minor, to previously existing language feaures and several new language features. The changes are summarized in Annex E. In addition, several VHDL library packages that were previously defined in separate standards are now defined in this standard, ensuring that they are treated as integral parts of the language. Finally, this revision incorporates the IEEE Property Specification Language (PSL) as part of VHDL. The combination of these changes significantly improves VHDL as a language for specification, design, and verification of complex electronic systems.

The maintenance of the VHDL language standard is an ongoing process. The chair of the VHDL Analysis and Standardization Group extends his gratitude to all who have participated in this revision, both in the IEEE committees and the Accellera effort, and encourages the participation of all interested parties in future language revisions.[b]

Notice to users

Laws and regulations

Users of these documents should consult all applicable laws and regulations. Compliance with the provisions of this standard does not imply compliance to any applicable regulatory requirements. Implementers of the standard are responsible for observing or referring to the applicable regulatory requirements. IEEE does not, by the publication of its standards, intend to urge action that is not in compliance with applicable laws, and these documents may not be construed as doing so.

Copyrights

This document is copyrighted by the IEEE. It is made available for a wide variety of both public and private uses. These include both use, by reference, in laws and regulations, and use in private self-regulation, standardization, and the promotion of engineering practices and methods. By making this document available for use and adoption by public authorities and private users, the IEEE does not waive any rights in copyright to this document.

[a]More information is available at www.accellera.org.

[b]If interested in participating, please contact the VASG at stds-vasg@ieee.org or visit: http://www.eda.org/vasg.

Updating of IEEE documents

Users of IEEE standards should be aware that these documents may be superseded at any time by the issuance of new editions or may be amended from time to time through the issuance of amendments, corrigenda, or errata. An official IEEE document at any point in time consists of the current edition of the document together with any amendments, corrigenda, or errata then in effect. In order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit the IEEE Standards Association website at http://ieeexplore.ieee.org/xpl/standards.jsp, or contact the IEEE at the address listed previously.

For more information about the IEEE Standards Association or the IEEE standards development process, visit the IEEE-SA website at http://standards.ieee.org.

Errata

Errata, if any, for this and all other standards can be accessed at the following URL: http://standards.ieee.org/reading/ieee/updates/errata/index.html. Users are encouraged to check this URL for errata periodically.

Interpretations

Current interpretations can be accessed at the following URL: http://standards.ieee.org/reading/ieee/interp/index.html.

Patents

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 or patent applications for which a license may be required to implement an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

Participants

At the time this standard was submitted to the IEEE-SA Standards Board for approval, the 1076 Working Group had the following membership:

Jim Lewis, *Chair*
Chuck Swart, *Vice Chair*
Peter Ashenden, *Secretary and Technical Editor*

John Aynsley
Stephen Bailey
Victor Berman
Patrick Bryant
Roland R. Henrie
Daniel Kho
Françoise Martinolle
Egert Molenkamp
Robert Myers
Farrell Ostler
John Ries
Timothy Schneider
Sukrit Shankar
John Shields
Alain Vachoux
Ajayharsh Varikat
Richard Wallace
John Willis
Alex Zamfirescu
Mark Zwolinski

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

Peter Ashenden
Bakul Banerjee
Victor Berman
Martin J. Bishop
Lyle Bullock
James Case
Keith Chow
Ernst Christen
S. Claassen
Kevin Coggins
Joanne Degroat
Thomas Dineen
Steven Dovich
George Economakos
Charles Gardiner
Sergiu Goma
Randall Groves
William Hanna
M. Hashmi
Werner Hoelzl
Joseph Hupcey
Piotr Karocki
Kurt Kermes
Jim Lewis
G. Luri
Timothy McBrayer
Donald Mills
Egbert Molenkamp
Robert Myers
Prajit Nair
Z. Navabi
Michael S. Newman
Ulrich Pohl
Bartien Sayogo
Timothy Schneider
Stephen Schwarm
John Sheppard
John Shields
David Smith
Walter Struppler
Chuck Swart
Lance Thompson
Yatin Trivedi
Alain Vachoux
Ajayharsh Varikat
Srinivasa Vemuru
Kathy Werner
John Willis
Paul Work
Mark Zwolinski

When the IEEE-SA Standards Board approved this standard on 26 September 2008, it had the following membership:

Robert M. Grow, *Chair*
Thomas Prevost, *Vice Chair*
Steve M. Mills, *Past Chair*
Judith Gorman, *Secretary*

Victor Berman
Richard DeBlasio
Andy Drozd
Mark Epstein
Alexander Gelman
William R. Goldbach
Arnold M. Greenspan
Kenneth S. Hanus
Jim Hughes
Richard H. Hulett
Young Kyun Kim
Joseph L. Koepfinger*
John Kulick
David J. Law
Glenn Parsons
Ronald C. Petersen
Chuck Powers
Narayanan Ramachandran
Jon Walter Rosdahl
Robby Robson
Anne-Marie Sahazizian
Malcolm V. Thaden
Howard L. Wolfman
Don Wright

*Member Emeritus

Also included are the following nonvoting IEEE-SA Standards Board liaisons:

Satish K. Aggarwal, *NRC Representative*
Michael Janezic, *NIST Representative*

Jennie Steinhagen
IEEE Standards Program Manager, Document Development

Michael D. Kipness
IEEE Standards Program Manager, Technical Program Development

Contents

IEEE Standard VHDL Language Reference Manual

1. Overview

1.1 Scope

This standard revises and enhances the VHDL language reference manual (LRM) by including a standard C language interface specification; specifications from previously separate, but related, standards IEEE Std 1164™-1993 [B16],[1] IEEE Std 1076.2™-1996 [B11], and IEEE Std 1076.3™-1997 [B12]; and general language enhancements in the areas of design and verification of electronic systems.

1.2 Purpose

The VHDL language was defined for use in the design and documentation of electronics systems. It is revised to incorporate capabilities that improve the language's usefulness for its intended purpose as well as extend it to address design verification methodologies that have developed in industry. These new design and verification capabilities are required to ensure VHDL remains relevant and valuable for use in electronic systems design and verification. Incorporation of previously separate, but related standards, simplifies the maintenance of the specifications.

[1]The numbers in brackets correspond to those of the bibliography in Annex J.

1.3 Structure and terminology of this standard

1.3.1 General

This standard is organized into clauses, each of which focuses on some particular area of the language. Within each clause, individual constructs or concepts are discussed in each subclause.

Each subclause describing a specific construct begins with an introductory paragraph. Next, the syntax of the construct is described using one or more grammatical *productions*.

A set of paragraphs describing the meaning and restrictions of the construct in narrative form then follow.

In this document, the word *shall* is used to indicate a mandatory requirement. The word *should* is used to indicate a recommendation. The word *may* is used to indicate a permissible action. The word *can* is used for statements of possibility and capability.

Finally, each clause may end with examples, notes, and references to other pertinent clauses.

1.3.2 Syntactic description

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

a) Lowercase words in roman font, some containing embedded underlines, are used to denote syntactic categories, for example:

    formal_port_list

   Whenever the name of a syntactic category is used, apart from the syntax rules themselves, spaces take the place of underlines [thus, “formal port list” would appear in the narrative description when referring to the syntactic category in item a)].

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

    array

   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, as follows:

    letter_or_digit ::= letter | digit
    choices ::= choice { | choice }

   In the first instance, an occurrence of “letter_or_digit” can be replaced by either “letter” or “digit.” In the second case, “choices” can be replaced by a list of “choice,” separated by vertical bars [see item f) for the meaning of braces].

e) Square brackets [ ] enclose optional items on the right-hand side of a production; thus, the following two productions are equivalent:

    return_statement ::= **return** [expression] ;
    return_statement ::= **return** ; | **return** expression ;

   Note, however, that the initial and terminal square brackets in the right-hand side of the production for signatures (see 4.5.3) are part of the syntax of signatures and do not indicate that the entire right-hand side is optional.

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. Thus, the following two productions are equivalent:

term ::= factor { multiplying_operator factor }
term ::= factor | term multiplying_operator factor

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, *type*_name and *subtype*_name are both syntactically equivalent to name alone.

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

1.3.3 Semantic description

The meaning and restrictions of a particular construct are described with a set of narrative rules immediately following the syntactic productions. In these rules, an italicized term indicates the definition of that term, and identifiers appearing entirely in uppercase letters refer to definitions in package STANDARD (see 16.3).

The following terms are used in these semantic descriptions with the following meanings:

erroneous: The condition described represents an ill-formed description; however, implementations are not required to detect and report this condition. Conditions are deemed erroneous only when it is impossible in general to detect the condition during the processing of the language.

error: The condition described represents an ill-formed description; implementations are required to detect the condition and report an error to the user of the tool.

illegal: A synonym for “error.”

legal: The condition described represents a well-formed description.

1.3.4 Front matter, examples, notes, references, and annexes

Prior to this subclause are several pieces of introductory material; following Clause 24 are some annexes and an index. The front matter, annexes (except Annex B), and index serve to orient and otherwise aid the user of this standard, but are not part of the definition of VHDL; Annex B, however, is normative.

Some clauses of this standard contain examples, notes, and cross-references to other clauses of the standard; these parts always appear at the end of a clause. Examples are meant to illustrate the possible forms of the construct described. Illegal examples are italicized. 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 this standard, notes are set as enumerated paragraphs in a font smaller than the rest of the text. Cross-references are meant to guide the user to other relevant clauses of the standard. Examples, notes, and cross-references are not part of the definition of the language.

1.3.5 Incorporation of Property Specification Language

VHDL incorporates the simple subset of the Property Specification Language (PSL) as an embedded language for formal specification of the behavior of a VHDL description. PSL is defined by IEEE Std 1850™-2005.[2] All PSL constructs that appear in a VHDL description shall conform to the

[2]Information on references can be found in Clause 2.

VHDL flavor of PSL. Within this standard, reference is made to syntactic rules of PSL. Each such reference has the italicized prefix *PSL_* and corresponds to the syntax rule in IEEE Std 1850-2005 with the same name but without the prefix.

2. Normative references

The following referenced documents are indispensable for the application of this document (i.e., they must be understood and used, so each referenced document is cited in text and its relationship to this document is explained). For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

IEEE Std 754™-1985 (Reaff 1990), IEEE Standard for Binary Floating-Point Arithmetic.[3, 4]

IEEE Std 854™-1987 (Reaff 1994), IEEE Standard for Radix-Independent Floating-Point Arithmetic.

IEEE Std 1850™-2005, IEEE Standard for Property Specification Language (PSL).

ISO/IEC 8859-1:1998, Information technology—8-bit single-byte coded graphic character sets—Part 1: Latin alphabet No. 1.[5]

ISO/IEC 9899:1999, Programming languages—C.

ISO/IEC 9899:1999/Cor 1:2001, Programming languages—C, Technical Corrigendum 1.

ISO/IEC 9899:1999/Cor 2:2004, Programming languages—C, Technical Corrigendum 2.

ISO/IEC 19501:2005, Information technology—Open Distributed Processing—Unified Modeling Language (UML) Version 1.4.2.

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

[4]The IEEE standards or products referred to in this clause are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

[5]ISO/IEC publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse (http://www.iso.ch/). ISO/IEC publications are also available in the United States from Global Engineering Documents, 15 Inverness Way East, Englewood, Colorado 80112, USA (http://global.ihs.com/). Electronic copies are available in the United States from the American National Standards Institute, 25 West 43rd Street, 4th Floor, New York, NY 10036, USA (http://www.ansi.org/).

3. Design entities and configurations

3.1 General

The *design entity* is the primary hardware abstraction in VHDL. It represents a portion of a hardware design that has well-defined inputs and outputs and performs a well-defined function. A design entity may represent an entire system, a subsystem, a board, a chip, a macro-cell, a logic gate, or any level of abstraction in-between. A *configuration* can be used to describe how design entities are put together to form a complete design.

A design entity may be described in terms of a hierarchy of *blocks*, each of which represents a portion of the whole design. The top-level block in such a hierarchy is the design entity itself; such a block is an *external* block that resides in a library and may be used as a component of other designs. Nested blocks in the hierarchy are *internal* blocks, defined by block statements (see 11.2).

A design entity may also be described in terms of interconnected components. Each component of a design entity may be bound to a lower-level design entity in order to define the structure or behavior of that component. Successive decomposition of a design entity into components, and binding those components to other design entities that may be decomposed in like manner, results in a hierarchy of design entities representing a complete design. Such a collection of design entities is called a *design hierarchy*. The bindings necessary to identify a design hierarchy can be specified in a configuration of the top-level entity in the hierarchy.

This clause describes the way in which design entities and configurations are defined. A design entity is defined by an *entity declaration* together with a corresponding *architecture body*. A configuration is defined by a *configuration declaration*.

3.2 Entity declarations

3.2.1 General

An entity declaration defines the interface between a given design entity and the environment in which it is used. It may also specify declarations and statements that are part of the design entity. A given entity declaration may be shared by many design entities, each of which has a different architecture. Thus, an entity declaration can potentially represent a class of design entities, each with the same interface.

entity_declaration ::=
    entity identifier **is**
        entity_header
        entity_declarative_part
    [**begin**
        entity_statement_part]
    end [**entity**] [*entity*_simple_name] ;

The entity header and entity declarative part consist of declarative items that pertain to each design entity whose interface is defined by the entity declaration. The entity statement part, if present, consists of concurrent statements that are present in each such design entity.

If a simple name appears at the end of an entity declaration, it shall repeat the identifier of the entity declaration.

3.2.2 Entity header

The entity header declares objects used for communication between a design entity and its environment.

entity_header ::=
    [*formal*_generic_clause]
    [*formal*_port_clause]

The generic list in the formal generic clause defines generics whose associated actuals may be determined by the environment (see 6.5.6.2). The port list in the formal port clause defines the input and output ports of the design entity (see 6.5.6.3).

In certain circumstances, the names of generics and ports declared in the entity header become visible outside of the design entity (see 12.2 and 12.3).

Examples:

— An entity declaration with port declarations only:

```
entity Full_Adder is
   port (X, Y, Cin: in Bit; Cout, Sum: out Bit);
end Full_Adder;
```

— An entity declaration with generic declarations also:

```
entity AndGate is
   generic (N: Natural := 2);
   port (Inputs: in Bit_Vector (1 to N);
         Result: out Bit);
end entity AndGate;
```

— An entity declaration with neither:

```
entity TestBench is
end TestBench;
```

3.2.3 Entity declarative part

The entity declarative part of a given entity declaration declares items that are common to all design entities whose interfaces are defined by the given entity declaration.

entity_declarative_part ::=
    { entity_declarative_item }

entity_declarative_item ::=
      subprogram_declaration
    | subprogram_body
    | subprogram_instantiation_declaration
    | package_declaration
    | package_body
    | package_instantiation_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | *shared*_variable_declaration
    | file_declaration
    | alias_declaration

| attribute_declaration
| attribute_specification
| disconnection_specification
| use_clause
| group_template_declaration
| group_declaration
| *PSL*_Property_Declaration
| *PSL*_Sequence_Declaration
| *PSL*_Clock_Declaration

Names declared by declarative items in the entity declarative part of a given entity declaration are visible within the bodies of corresponding design entities, as well as within certain portions of a corresponding configuration declaration.

The various kinds of declaration are described in Clause 6, and the various kinds of specification are described in Clause 7. The use clause, which makes externally defined names visible within the block, is described in Clause 12.

Example:

— An entity declaration with entity declarative items:

```
entity ROM is
   port (Addr: in  Word;
         Data: out Word;
         Sel:  in  Bit);
   type Instruction is array (1 to 5) of Natural;
   type Program is array (Natural range <>) of Instruction;
   use Work.OpCodes.all, Work.RegisterNames.all;
   constant ROM_Code: Program :=
      (
         (STM, R14, R12, 12, R13),
         (LD,  R7,  32,  0,  R1 ),
         (BAL, R14, 0,   0,  R7 ),
               .
               .    --  etc.
               .
      ) ;
end ROM;
```

NOTE—The entity declarative part of a design entity whose corresponding architecture is decorated with the 'FOREIGN attribute is subject to special elaboration rules. See 14.4.1.[6]

3.2.4 Entity statement part

The entity statement part contains concurrent statements that are common to each design entity with this interface.

entity_statement_part ::=
    { entity_statement }

entity_statement ::=
     concurrent_assertion_statement
    | *passive*_concurrent_procedure_call_statement

[6]Notes in text, tables, and figures are given for information only and do not contain requirements needed to implement the standard.

| *passive*_process_statement
| *PSL*_PSL_Directive

It is an error if any statements other than concurrent assertion statements, concurrent procedure call statements, process statements, or PSL directives appear in the entity statement part. All entity statements shall be passive (see 11.3). Such statements may be used to monitor the operating conditions or characteristics of a design entity.

Example:

— An entity declaration with statements:

```
entity Latch is
   port (Din:  in  Word;
         Dout: out Word;
         Load: in  Bit;
         Clk:  in  Bit );
   constant Setup: Time := 12 ns;
   constant PulseWidth: Time := 50 ns;
   use Work.TimingMonitors.all;
begin
   assert Clk='1' or Clk'Delayed'Stable (PulseWidth);
   CheckTiming (Setup, Din, Load, Clk);
end;
```

NOTE—The entity statement part of a design entity whose corresponding architecture is decorated with the 'FOREIGN attribute is subject to special elaboration rules. See 14.5.1.

3.3 Architecture bodies

3.3.1 General

An architecture body defines the body of a design entity. It specifies the relationships between the inputs and outputs of a design entity and may be expressed in terms of structure, dataflow, or behavior. Such specifications may be partial or complete.

architecture_body ::=
architecture identifier **of** *entity*_name **is**
architecture_declarative_part
begin
architecture_statement_part
end [**architecture**] [*architecture*_simple_name] ;

The identifier defines the simple name of the architecture body; this simple name distinguishes architecture bodies associated with the same entity declaration.

The entity name identifies the name of the entity declaration that defines the interface of this design entity. For a given design entity, both the entity declaration and the associated architecture body shall reside in the same library.

If a simple name appears at the end of an architecture body, it shall repeat the identifier of the architecture body.

More than one architecture body may exist corresponding to a given entity declaration. Each declares a different body with the same interface; thus, each together with the entity declaration represents a different design entity with the same interface.

NOTE—Two architecture bodies that are associated with different entity declarations may have the same simple name, even if both architecture bodies (and the corresponding entity declarations) reside in the same library.

3.3.2 Architecture declarative part

The architecture declarative part contains declarations of items that are available for use within the block defined by the design entity.

architecture_declarative_part ::=
    { block_declarative_item }

block_declarative_item ::=
      subprogram_declaration
    | subprogram_body
    | subprogram_instantiation_declaration
    | package_declaration
    | package_body
    | package_instantiation_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | *shared*_variable_declaration
    | file_declaration
    | alias_declaration
    | component_declaration
    | attribute_declaration
    | attribute_specification
    | configuration_specification
    | disconnection_specification
    | use_clause
    | group_template_declaration
    | group_declaration
    | *PSL*_Property_Declaration
    | *PSL*_Sequence_Declaration
    | *PSL*_Clock_Declaration

The various kinds of declaration are described in Clause 6, and the various kinds of specification are described in Clause 7. The use clause, which makes externally defined names visible within the block, is described in Clause 12.

NOTE—The declarative part of an architecture decorated with the 'FOREIGN attribute is subject to special elaboration rules. See 14.4.1.

3.3.3 Architecture statement part

The architecture statement part contains statements that describe the internal organization and/or operation of the block defined by the design entity.

architecture_statement_part ::=
    { concurrent_statement }

All of the statements in the architecture statement part are concurrent statements, which execute asynchronously with respect to one another. The various kinds of concurrent statements are described in Clause 11.

Examples:

— A body of entity Full_Adder:

```
architecture DataFlow of Full_Adder is
   signal A,B: Bit;
begin
   A <= X xor Y;
   B <= A and Cin;
   Sum <= A xor Cin;
   Cout <= B or (X and Y);
end architecture DataFlow;
```

— A body of entity TestBench:

```
library Test;
use Test.Components.all;
architecture Structure of TestBench is
   component Full_Adder
      port (X, Y, Cin: Bit; Cout, Sum: out Bit);
   end component;
signal A,B,C,D,E,F,G: Bit;
   signal OK: Boolean;
begin
   UUT:        Full_Adder port map (A,B,C,D,E);
   Generator:  AdderTest  port map (A,B,C,F,G);
   Comparator: AdderCheck port map (D,E,F,G,OK);
end Structure;
```

— A body of entity AndGate:

```
architecture Behavior of AndGate is
begin
   process (Inputs)
      variable Temp: Bit;
   begin
      Temp := '1';
      for i in Inputs'Range loop
         if Inputs(i) = '0' then
            Temp := '0';
            exit;
         end if;
      end loop;
      Result <= Temp after 10 ns;
   end process;
end Behavior;
```

NOTE—The statement part of an architecture decorated with the 'FOREIGN attribute is subject to special elaboration rules. See 14.5.1.

3.4 Configuration declarations

3.4.1 General

The binding of component instances to design entities is performed by configuration specifications (see 7.3); such specifications appear in the declarative part of the block in which the corresponding component instances are created. In certain cases, however, it may be appropriate to leave unspecified the binding of component instances in a given block and to defer such specification until later. A configuration declaration provides the mechanism for specifying such deferred bindings.

configuration_declaration ::=
    configuration identifier **of** *entity*_name **is**
        configuration_declarative_part
        { verification_unit_binding_indication ; }
        block_configuration
    end [**configuration**] [*configuration*_simple_name] ;

configuration_declarative_part ::=
    { configuration_declarative_item }

configuration_declarative_item ::=
      use_clause
    | attribute_specification
    | group_declaration

The entity name identifies the name of the entity declaration that defines the design entity at the root of the design hierarchy. For a configuration of a given design entity, both the configuration declaration and the corresponding entity declaration shall reside in the same library.

If a simple name appears at the end of a configuration declaration, it shall repeat the identifier of the configuration declaration.

A verification unit binding indication in a configuration declaration binds one or more PSL verification units to the design entity at the root of the design hierarchy. Verification unit binding indications are described in 7.3.4.

NOTE 1—A configuration declaration achieves its effect entirely through elaboration (see Clause 14). There are no behavioral semantics associated with a configuration declaration.

NOTE 2—A given configuration may be used in the definition of another, more complex configuration.

Examples:

— An architecture of a microprocessor:

```
architecture Structure_View of Processor is
   component ALU port ( ··· ); end component;
   component MUX port ( ··· ); end component;
   component Latch port ( ··· ); end component;
begin
   A1: ALU port map ( ··· );
   M1: MUX port map ( ··· );
   M2: MUX port map ( ··· );
   M3: MUX port map ( ··· );
   L1: Latch port map ( ··· );
   L2: Latch port map ( ··· );
end Structure_View;
```

— A configuration of the microprocessor:

```
library TTL, Work;
configuration V4_27_87 of Processor is
   use Work.all;
   for Structure_View
      for A1: ALU
         use configuration TTL.SN74LS181;
      end for;
      for M1,M2,M3: MUX
         use entity Multiplex4 (Behavior);
      end for;
      for all: Latch
         -- use defaults
      end for;
   end for;
end configuration V4_27_87;
```

3.4.2 Block configuration

A block configuration defines the configuration of a block. Such a block is either an internal block defined by a block statement or an external block defined by a design entity. If the block is an internal block, the defining block statement is either an explicit block statement or an implicit block statement that is itself defined by a generate statement.

block_configuration ::=
    for block_specification
        { use_clause }
        { configuration_item }
    end for ;

block_specification ::=
      *architecture*_name
    | *block_statement*_label
    | *generate_statement*_label [(generate_specification)]

generate_specification ::=
      *static*_discrete_range
    | *static*_expression
    | *alternative*_label

configuration_item ::=
      block_configuration
    | component_configuration

The block specification identifies the internal or external block to which this block configuration applies.

If a block configuration appears immediately within a configuration declaration, then the block specification of that block configuration shall be an architecture name, and that architecture name shall denote a design entity body whose interface is defined by the entity declaration denoted by the entity name of the enclosing configuration declaration.

If a block configuration appears immediately within a component configuration, then the corresponding components shall be fully bound (see 7.3.2.2), the block specification of that block configuration shall be an

architecture name, and that architecture name shall denote the same architecture body as that to which the corresponding components are bound.

If a block configuration appears immediately within another block configuration, then the block specification of the contained block configuration shall be a block statement or generate statement label, and the label shall denote a block statement or generate statement that is contained immediately within the block denoted by the block specification of the containing block configuration.

If the scope of a declaration (see 12.2) includes the end of the declarative part of a block corresponding to a given block configuration, then the scope of that declaration extends to each configuration item contained in that block configuration, with the exception of block configurations that configure external blocks. Similarly, if a declaration is visible (either directly or by selection) at the end of the declarative part of a block corresponding to a given block configuration, then the declaration is visible in each configuration item contained in that block configuration, with the exception of block configurations that configure external blocks. Additionally, if a given declaration is a homograph of a declaration that a use clause in the block configuration makes potentially directly visible, then the given declaration is not directly visible in the block configuration or any of its configuration items. See 12.3.

For any name that is the label of a block statement appearing immediately within a given block, a corresponding block configuration may appear as a configuration item immediately within a block configuration corresponding to the given block. For any collection of names that are labels of instances of the same component appearing immediately within a given block, a corresponding component configuration may appear as a configuration item immediately within a block configuration corresponding to the given block.

For any name that is the label of a generate statement immediately within a given block, one or more corresponding block configurations may appear as configuration items immediately within a block configuration corresponding to the given block. Such block configurations apply to implicit blocks generated by that generate statement. If such a block configuration contains a generate specification that is a static discrete range, then the block configuration applies to those implicit block statements that are generated for the specified range of values of the corresponding generate parameter; the discrete range has no significance other than to define the set of generate statement parameter values implied by the discrete range. If such a block configuration contains a generate specification that is a static expression, then the block configuration applies only to the implicit block statement generated for the specified value of the corresponding generate parameter. If such a block configuration contains a generate specification that is an alternative label, then the block configuration applies only to the implicit block generated for the generate statement body following the alternative label in the generate statement, if and only if the condition after the alternative label evaluates to TRUE (for an if generate statement) or the case generate alternative containing the alternative label is the chosen alternative (for a case generate statement). If no generate specification appears in such a block configuration, then it applies to exactly one of the following sets of blocks:

— All implicit blocks (if any) generated by the corresponding generate statement, if and only if the corresponding generate statement is a for generate statement.

— The implicit block generated by the corresponding generate statement, if and only if the corresponding generate statement is an if generate statement and if the first condition after **if** evaluates to TRUE.

— No implicit or explicit blocks, if and only if the corresponding generate statement is an if generate statement and the first condition after **if** evaluates to FALSE.

If the block specification of a block configuration contains a generate statement label, and if this label contains a generate specification, then:

— If the generate specification is a discrete range or an expression, then it is an error if the generate statement denoted by the generate statement label is not a for generate statement. Moreover, for a generate specification that is a discrete range, it is an error if the type of the discrete range is not the

same as the type of the discrete range of the generate parameter specification and if any value in the range does not belong to the discrete range of the generate parameter specification. Similarly, for a generate specification that is an expression, it is an error if the type of the expression is not the same as the type of the discrete range of the generate parameter specification and if the value of the expression does not belong to the discrete range of the generate parameter specification.

— If the generate specification is an alternative label, then it is an error if the generate statement denoted by the generate statement label is not an if generate statement that includes the alternative label or a case generate statement that includes the alternative label.

If the block specification of a block configuration contains a generate statement label that denotes an if generate statement, and if the first condition after **if** has an alternative label, then it is an error if the generate statement label does not contain a generate specification that is an alternative label. Similarly, if the block specification of a block configuration contains a generate statement label that denotes a case generate statement, then it is an error if the generate statement label does not contain a generate specification that is an alternative label.

Within a given block configuration, whether implicit or explicit, an implicit block configuration is assumed to appear for any block statement that appears within the block corresponding to the given block configuration, if no explicit block configuration appears for that block statement. Similarly, an implicit component configuration is assumed to appear for each component instance that appears within the block corresponding to the given block configuration, if no explicit component configuration appears for that instance. Such implicit configuration items are assumed to appear following all explicit configuration items in the block configuration.

It is an error if, in a given block configuration, more than one configuration item is defined for the same block or component instance.

NOTE 1—As a result of the rules described in the preceding paragraphs and in Clause 12, a simple name that is visible by selection at the end of the declarative part of a given block is also visible by selection within any configuration item contained in a corresponding block configuration. If such a name is directly visible at the end of the given block declarative part, it will likewise be directly visible in the corresponding configuration items, unless a use clause for a different declaration with the same simple name appears in the corresponding configuration declaration, and the scope of that use clause encompasses all or part of those configuration items. If such a use clause appears, then the name will be directly visible within the corresponding configuration items except at those places that fall within the scope of the additional use clause (at which places neither name will be directly visible).

NOTE 2—If an implicit configuration item is assumed to appear within a block configuration, that implicit configuration item will never contain explicit configuration items.

NOTE 3—If the block specification in a block configuration specifies a generate statement label, and if this label contains a generate specification that is a discrete range, then the direction specified or implied by the discrete range has no significance other than to define, together with the bounds of the range, the set of generate statement parameter values denoted by the range. Thus, the following two block configurations are equivalent:

```
for Adders(31 downto 0) ··· end for;
for Adders(0 to 31) ··· end for;
```

NOTE 4—A block configuration is allowed to appear immediately within a configuration declaration only if the entity declaration denoted by the entity name of the enclosing configuration declaration has associated architectures. Furthermore, the block specification of the block configuration shall denote one of these architectures.

Examples:

— A block configuration for a design entity:

```
for ShiftRegStruct                          --  An architecture name.
   --  Configuration items
   --  for blocks and components
   --  within ShiftRegStruct.
end for;
```

— A block configuration for a block statement:

```
for B1                                      --  A block label.
   --  Configuration items
   --  for blocks and components
   --  within block B1.
end for;
```

3.4.3 Component configuration

A component configuration defines the configuration of one or more component instances in a corresponding block.

component_configuration ::=
    for component_specification
        [binding_indication ;]
        { verification_unit_binding_indication ; }
        [block_configuration]
    end for ;

The component specification (see 7.3) identifies the component instances to which this component configuration applies. A component configuration that appears immediately within a given block configuration applies to component instances that appear immediately within the corresponding block.

It is an error if two component configurations apply to the same component instance.

If the component configuration contains a binding indication (see 7.3.2), then the component configuration implies a configuration specification for the component instances to which it applies. This implicit configuration specification has the same component specification and binding indication as that of the component configuration.

If a given component instance is unbound in the corresponding block, then any explicit component configuration for that instance that does not contain an explicit binding indication will contain an implicit, default binding indication (see 7.3.3). Similarly, if a given component instance is unbound in the corresponding block, then any implicit component configuration for that instance will contain an implicit, default binding indication.

A verification unit binding indication in a configuration declaration binds one or more PSL verification units to the instance of the design entity bound to the component instances identified by the component specification. Verification unit binding indications are described in 7.3.4.

It is an error if a component configuration contains an explicit block configuration and the component configuration does not bind all identified component instances to the same design entity.

Within a given component configuration, whether implicit or explicit, an implicit block configuration is assumed for the design entity to which the corresponding component instance is bound, if no explicit block configuration appears and if the corresponding component instance is fully bound.

Examples:

— A component configuration with binding indication:

```
for all: IOPort
   use entity StdCells.PadTriState4 (DataFlow)
      port map (Pout=>A, Pin=>B, IO=>Dir, Vdd=>Pwr, Gnd=>Gnd);
end for;
```

— A component configuration containing block configurations:

```
for D1: DSP
   for DSP_STRUCTURE
      --  Binding specified in design entity or else defaults.
      for Filterer
         --  Configuration items for filtering components.
      end for;
      for Processor
         --  Configuration items for processing components.
      end for;
   end for;
end for;
```

NOTE—The requirement that all component instances corresponding to a block configuration be bound to the same design entity makes the following configuration illegal:

```
architecture A of E is
   component C is end component C;
   for L1: C use entity E1(X);
   for L2: C use entity E2(X);
begin
   L1: C;
   L2: C;
end architecture A;


configuration Illegal of Work.E is
   for A
      for all: C
         for X        --  Does not apply to the same design entity in all instances of C.
            ...
         end for;     -- X
      end for; -- C
   end for; -- A
end configuration Illegal;
```


4. Subprograms and packages

4.1 General

Subprograms define algorithms for computing values or exhibiting behavior. They may be used as computational resources to convert between values of different types, to define the resolution of output values driving a common signal, or to define portions of a process. Packages provide a means of defining these and other resources in a way that allows different design units or different parts of a given design unit to share the same declarations.

There are two forms of subprograms: procedures and functions. A procedure call is a statement; a function call is an expression and returns a value. Certain functions, designated *pure* functions, return the same value each time they are called with the same values as actual parameters; the remainder, impure functions, may return a different value each time they are called, even when multiple calls have the same actual parameter values. In addition, *impure* functions can update objects outside of their scope and can access a broader class of values than can pure functions. The definition of a subprogram can be given in two parts: a subprogram declaration defining its calling conventions, and a subprogram body defining its execution.

Packages may also be defined in two parts. A package declaration defines the visible contents of a package; a package body provides hidden details. In particular, a package body contains the bodies of any subprograms declared in the package declaration.

4.2 Subprogram declarations

4.2.1 General

A subprogram declaration declares a procedure or a function, as indicated by the appropriate reserved word.

```
subprogram_declaration ::=
    subprogram_specification ;

subprogram_specification ::=
    procedure_specification | function_specification

procedure_specification ::=
    procedure designator
        subprogram_header
        [ [ parameter ] ( formal_parameter_list ) ]

function_specification ::=
    [ pure | impure ] function designator
        subprogram_header
        [ [ parameter ] ( formal_parameter_list ) ] return type_mark

subprogram_header ::=
    [ generic ( generic_list )
    [ generic_map_aspect ] ]

designator ::= identifier | operator_symbol

operator_symbol ::= string_literal
```

The specification of a procedure specifies its designator, its generics (if any), and its formal parameters (if any). The specification of a function specifies its designator, its generics (if any), its formal parameters (if any), the subtype of the returned value (the *result subtype*), and whether or not the function is pure. A function is *impure* if its specification contains the reserved word **impure**; otherwise, it is said to be *pure*. A procedure designator is always an identifier. A function designator is either an identifier or an operator symbol. A designator that is an operator symbol is used for the overloading of an operator (see 4.5.2). The sequence of characters represented by an operator symbol shall be an operator belonging to one of the classes of operators defined in 9.2. Extra spaces are not allowed in an operator symbol, and the case of letters is not significant.

If the subprogram header is empty, the subprogram declared by a subprogram declaration is called a *simple subprogram*. If the subprogram header contains the reserved word **generic**, a generic list, and no generic map aspect, the subprogram is called an *uninstantiated subprogram*. If the subprogram header contains the reserved word **generic**, a generic list, and a generic map aspect, the subprogram is called a *generic-mapped subprogram*. A subprogram declared with a generic list in which every generic declaration has a default, and with no generic map aspect, is considered to be an uninstantiated subprogram, not a generic-mapped subprogram with default associations for all of the generic declarations. A generic list in a subprogram declaration is equivalent to a generic clause containing that generic list (see 6.5.6.2).

An uninstantiated subprogram shall not be called, except as a recursive call within the body of the uninstantiated subprogram. Moreover, an uninstantiated subprogram shall not be used as a resolution function or used as a conversion function in an association list.

It is an error if the result subtype of a function denotes either a file type or a protected type. Moreover, it is an error if the result subtype of a pure function denotes an access type or a subtype that has a subelement of an access type.

NOTE 1—All subprograms can be called recursively. In the case of an instantiated subprogram, a reference to the uninstantiated subprogram within the uninstantiated subprogram is interpreted as a reference to the instance (see 4.4). Hence, the subprogram can be called recursively using the name of the uninstantiated subprogram. The effect is a recursive call of the instance.

NOTE 2—The restrictions on pure functions are enforced even when the function appears within a protected type. That is, pure functions whose body appears in the protected type body shall not directly reference variables declared immediately within the declarative region associated with the protected type. However, impure functions and procedures whose bodies appear in the protected type body may make such references. Such references are made only when the referencing subprogram has exclusive access to the declarative region associated with the protected type.

NOTE 3—The rule stating equivalence of a generic list in a subprogram header to a generic clause containing the generic list ensures that the generic list conforms to the same rules as a generic clause. A subprogram header is not defined to contain a generic clause directly, since that would introduce a semicolon into the syntax of a subprogram header.

4.2.2 Formal parameters

4.2.2.1 Formal parameter lists

The formal parameter list in a subprogram specification defines the formal parameters of the subprogram.

formal_parameter_list ::= *parameter*_interface_list

Formal parameters of subprograms may be constants, variables, signals, or files. In the first three cases, the mode of a parameter determines how a given formal parameter is accessed within the subprogram. The mode of a formal parameter, together with its class, also determines how such access is implemented. In the fourth case, that of files, the parameters have no mode.

For those parameters with modes, the only modes that are allowed for formal parameters of a procedure are **in**, **inout**, and **out**. If the mode is **in** and no object class is explicitly specified, **constant** is assumed. If the mode is **inout** or **out**, and no object class is explicitly specified, **variable** is assumed.

For those parameters with modes, the only mode that is allowed for formal parameters of a function is the mode **in** (whether this mode is specified explicitly or implicitly). The object class shall be **constant**, **signal**, or **file**. If no object class is explicitly given, **constant** is assumed.

In a subprogram call, the actual designator (see 6.5.7.1) associated with a formal parameter of class **signal** shall be a name denoting a signal. The actual designator associated with a formal of class **variable** shall be a name denoting a variable. The actual designator associated with a formal of class **constant** shall be an expression. The actual designator associated with a formal of class **file** shall be a name denoting a file.

NOTE—Attributes of an actual are never passed into a subprogram. References to an attribute of a formal parameter are legal only if that formal has such an attribute. Such references retrieve the value of the attribute associated with the formal.

4.2.2.2 Constant and variable parameters

For parameters of class **constant** or **variable**, only the values of the actual or formal are transferred into or out of the subprogram call. The manner of such transfers, and the accompanying access privileges that are granted for constant and variable parameters, are described in this subclause.

For a nonforeign subprogram having a parameter of a scalar type or an access type, or for a subprogram decorated with the 'FOREIGN attribute defined in package STANDARD for which the attribute value is of the form described in 20.2.4, the parameter is passed by copy. At the start of each call, if the mode is **in** or **inout**, the value of the actual parameter is copied into the associated formal parameter; it is an error if, after applying any conversion function or type conversion present in the actual part of the applicable association element (see 6.5.7.1), the value of the actual parameter does not belong to the subtype denoted by the subtype indication of the formal. After completion of the subprogram body, if the mode is **inout** or **out** and the associated actual parameter is not forced, the value of the formal parameter is copied back into the associated actual parameter; it is similarly an error if, after applying any conversion function or type conversion present in the formal part of the applicable association element, the value of the formal parameter does not belong to the subtype denoted by the subtype indication of the actual.

For a nonforeign subprogram having a parameter whose type is an array or record, an implementation may pass parameter values by copy, as for scalar types. In that case, after completion of the subprogram body, if the mode is **inout** or **out**, the value of each subelement of the formal parameter is only copied back to the corresponding subelement of the associated actual parameter if the subelement of the associated actual parameter is not forced. If a parameter of mode **out** is passed by copy, then the range of each index position of the actual parameter is copied in, and likewise for its subelements or slices. Alternatively, an implementation may achieve these effects by reference; that is, by arranging that every use of the formal parameter (to read or update its value) be treated as a use of the associated actual parameter throughout the execution of the subprogram call. The language does not define which of these two mechanisms is to be adopted for parameter passing, nor whether different calls to the same subprogram are to use the same mechanism. The execution of a subprogram is erroneous if its effect depends on which mechanism is selected by the implementation.

For a subprogram having a parameter whose type is a protected type, the parameter is passed by reference. It is an error if the mode of the parameter is other than **inout**.

For a formal parameter of a composite subtype, the index ranges of the formal, if it is an array, and of any array subelements, are determined as specified in 5.3.2.2. For a formal parameter of mode **in** or **inout**, it is an error if the value of the associated actual parameter (after application of any conversion function or type conversion present in the actual part) does not contain a matching subelement for each subelement of the

formal. It is also an error if the value of each subelement of the actual (after applying any conversion function or type conversion present in the actual part) does not belong to the subtype of the corresponding subelement of the formal. If the formal parameter is of mode **out** or **inout**, it is also an error if, at the end of the subprogram call, the value of each subelement of the formal (after applying any conversion function or type conversion present in the formal part) does not belong to the subtype of the corresponding subelement of the actual.

NOTE 1—For parameters of array and record types, the parameter passing rules imply that if no actual parameter of such a type is accessible by more than one path, then the effect of a subprogram call is the same whether or not the implementation uses copying for parameter passing. If, however, there are multiple access paths to such a parameter (for example, if another formal parameter is associated with the same actual parameter), then the value of the formal is undefined after updating the actual other than by updating the formal. A description using such an undefined value is erroneous.

NOTE 2—The value of an actual associated with a formal variable parameter of mode **out** is not copied into the formal parameter. Rather, the formal parameter is initialized based on its declared type, regardless of whether the implementation chooses to pass the parameter by copy or by reference. When a formal variable parameter of mode **out** is read, the current value of the formal parameter is read.

4.2.2.3 Signal parameters

For a formal parameter of class **signal**, references to the signal, the driver of the signal, or both, are passed into the subprogram call.

For a signal parameter of mode **in** or **inout**, the actual signal is associated with the corresponding formal signal parameter at the start of each call. Thereafter, during the execution of the subprogram body, a reference to the formal signal parameter within an expression is equivalent to a reference to the actual signal.

It is an error if signal-valued attributes 'STABLE, 'QUIET, 'TRANSACTION, and 'DELAYED of formal signal parameters of any mode are read within a subprogram.

A process statement contains a driver for each actual signal associated with a formal signal parameter of mode **out** or **inout** in a subprogram call. Similarly, a subprogram contains a driver for each formal signal parameter of mode **out** or **inout** declared in its subprogram specification.

For a signal parameter of mode **inout** or **out**, the driver of an actual signal is associated with the corresponding driver of the formal signal parameter at the start of each call. Thereafter, during the execution of the subprogram body, an assignment to the driver of a formal signal parameter is equivalent to an assignment to the driver of the actual signal.

If an actual signal is associated with a signal parameter of any mode, the actual shall be denoted by a static signal name. It is an error if a conversion function or type conversion appears in either the formal part or the actual part of an association element that associates an actual signal with a formal signal parameter.

If an actual signal is associated with a signal parameter of mode **in** or **inout**, and if the type of the formal is a scalar type, then it is an error if the subtype of the actual is not compatible with the subtype of the formal. Similarly, if an actual signal is associated with a signal parameter of mode **out** or **inout**, and if the type of the actual is a scalar type, then it is an error if the subtype of the formal is not compatible with the subtype of the actual.

For a formal parameter of a composite subtype, the index ranges of the formal, if it is an array, and of any array subelements, are determined as specified in 5.3.2.2. It is an error if the actual signal does not contain a matching subelement for each subelement of the formal. It is also an error if the mode of the formal is **in** or **inout** and if the value of each scalar subelement of the actual does not belong to the subtype of the corresponding subelement of the formal.

A formal signal parameter is a guarded signal if and only if it is associated with an actual signal that is a guarded signal. It is an error if the declaration of a formal signal parameter includes the reserved word **bus** (see 6.5.2).

NOTE—It is a consequence of the preceding rules that a procedure with an **out** or **inout** signal parameter called by a process does not have to complete in order for any assignments to that signal parameter within the procedure to take effect. Assignments to the driver of a formal signal parameter are equivalent to assignments directly to the actual driver contained in the process calling the procedure.

4.2.2.4 File parameters

For parameters of class **file**, references to the actual file are passed into the subprogram. No particular parameter-passing mechanism is defined by the language, but a reference to the formal parameter shall be equivalent to a reference to the actual parameter. It is an error if an association element associates an actual with a formal parameter of a file type and that association element contains a conversion function or type conversion. It is also an error if a formal of a file type is associated with an actual that is not of a file type.

At the beginning of a given subprogram call, a file parameter is open (see 5.5.2) if and only if the actual file object associated with the given parameter in a given subprogram call is also open. Similarly, at the beginning of a given subprogram call, both the access mode of and external file associated with (see 5.5.2) an open file parameter are the same as, respectively, the access mode of and the external file associated with the actual file object associated with the given parameter in the subprogram call.

At the completion of the execution of a given subprogram call, the actual file object associated with a given file parameter is open if and only if the formal parameter is also open. Similarly, at the completion of the execution of a given subprogram call, the access mode of and the external file associated with an open actual file object associated with a given file parameter are the same as, respectively, the access mode of and the external file associated with the associated formal parameter.

4.3 Subprogram bodies

A subprogram body specifies the execution of a subprogram.

```
subprogram_body ::=
    subprogram_specification is
        subprogram_declarative_part
    begin
        subprogram_statement_part
    end [ subprogram_kind ] [ designator ] ;

subprogram_declarative_part ::=
    { subprogram_declarative_item }

subprogram_declarative_item ::=
      subprogram_declaration
    | subprogram_body
    | subprogram_instantiation_declaration
    | package_declaration
    | package_body
    | package_instantiation_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
```

| file_declaration
| alias_declaration
| attribute_declaration
| attribute_specification
| use_clause
| group_template_declaration
| group_declaration

subprogram_statement_part ::=
{ sequential_statement }

subprogram_kind ::= **procedure** | **function**

The declaration of a subprogram is optional. In the absence of such a declaration, the subprogram specification of the subprogram body acts as the declaration. For each subprogram declaration, there shall be a corresponding body. If both a declaration and a body are given, the subprogram specification of the body shall lexically conform (see 4.10) to the subprogram specification of the declaration. Furthermore, both the declaration and the body shall occur immediately within the same declarative region (see 12.1).

If a subprogram kind appears at the end of a subprogram body, it shall repeat the reserved word given in the subprogram specification. If a designator appears at the end of a subprogram body, it shall repeat the designator of the subprogram.

It is an error if a variable declaration in a subprogram declarative part declares a shared variable. (See 6.4.2.4.)

A *foreign subprogram* is one that is decorated with the attribute 'FOREIGN, defined in package STANDARD (see 16.3). The STRING value of the attribute may specify implementation-dependent information about the foreign subprogram. Foreign subprograms may have non-VHDL implementations. An implementation may place restrictions on the appearance of a generic list and a generic map aspect in the declaration of a foreign subprogram. An implementation may also place restrictions on the allowable modes, classes, and types of the formal parameters to a foreign subprogram; such restrictions may include restrictions on the number and allowable order of the parameters.

Excepting foreign subprograms, the algorithm performed by a subprogram is defined by the sequence of statements that appears in the subprogram statement part. For a foreign subprogram, the algorithm performed is implementation defined.

The execution of a subprogram body, other than an uninstantiated subprogram body, is invoked by a subprogram call. For this execution, after establishing the association between the formal and actual parameters, the sequence of statements of the body is executed if the subprogram is not a foreign subprogram; otherwise, an implementation-defined action occurs. Upon completion of the body or implementation-dependent action, if exclusive access to an object of a protected type was granted during elaboration of the declaration of the subprogram (see 14.6), the exclusive access is rescinded. Then, return is made to the caller (and any necessary copying back of formal to actual parameters occurs).

A process or a subprogram is said to be a *parent* of a given subprogram S if that process or subprogram contains a procedure call or function call for S or for a parent of S. An instantiated subprogram is a parent of a given subprogram S if the uninstantiated subprogram of which the instantiated subprogram is an instance is a parent of S.

An *explicit* signal is a signal other than an implicit signal GUARD and other than one of the implicit signals defined by the predefined attributes 'DELAYED, 'STABLE, 'QUIET, or 'TRANSACTION. The *explicit ancestor* of an implicit signal is found as follows. The implicit signal GUARD has no explicit ancestor. An

explicit ancestor of an implicit signal defined by the predefined attributes 'DELAYED, 'STABLE, 'QUIET, or 'TRANSACTION is the signal found by recursively examining the prefix of the attribute. If the prefix denotes an explicit signal, a slice, or a member (see Clause 5) of an explicit signal, then that is the explicit ancestor of the implicit signal. Otherwise, if the prefix is one of the implicit signals defined by the predefined attributes 'DELAYED, 'STABLE, 'QUIET, or 'TRANSACTION, this rule is recursively applied. If the prefix is an implicit signal GUARD, then the signal has no explicit ancestor.

If a pure function subprogram is a parent of a given procedure and if that procedure contains a reference to an explicitly declared signal or variable object, or a slice or subelement (or slice thereof) of an explicit signal, then that object shall be declared within the declarative region formed by the function (see 12.1) or within the declarative region formed by the procedure; this rule also holds for the explicit ancestor, if any, of an implicit signal and also for the implicit signal GUARD. If a pure function is the parent of a given procedure, then that procedure shall not contain a reference to an explicitly declared file object (see 6.4.2.5) or to a shared variable (see 6.4.2.4).

Similarly, if a pure function subprogram contains a reference to an explicitly declared signal or variable object, or a slice or subelement (or slice thereof) of an explicit signal, then that object shall be declared within the declarative region formed by the function; this rule also holds for the explicit ancestor, if any, of an implicit signal and also for the implicit signal GUARD. A pure function shall not contain a reference to an explicitly declared file object.

A pure function shall not be the parent of an impure function.

The rules of the preceding three paragraphs apply to all pure function subprograms. For pure functions that are not foreign subprograms, violations of any of these rules are errors. However, since implementations cannot in general check that such rules hold for pure function subprograms that are foreign subprograms, a description calling pure foreign function subprograms not adhering to these rules is erroneous.

Example:

— The declaration of a foreign function subprogram:

```
package P is
   function F return INTEGER;
   attribute FOREIGN of F: function is
      "implementation-dependent information";
end package P;
```

NOTE 1—It follows from the visibility rules that a subprogram declaration shall be given if a call of the subprogram occurs textually before the subprogram body, and that such a declaration shall occur before the call itself.

NOTE 2—The preceding rules concerning pure function subprograms, together with the fact that function parameters shall be of mode **in**, imply that a pure function has no effect other than the computation of the returned value. Thus, a pure function invoked explicitly as part of the elaboration of a declaration, or one invoked implicitly as part of the simulation cycle, is guaranteed to have no effect on other objects in the description.

NOTE 3—VHDL does not define the parameter-passing mechanisms for foreign subprograms.

NOTE 4—The declarative parts and statement parts of subprograms decorated with the 'FOREIGN attribute are subject to special elaboration rules. See 14.4.1 and 14.6.

NOTE 5—A pure function subprogram shall not reference a shared variable. This prohibition exists because a shared variable cannot be declared in a subprogram declarative part and a pure function cannot reference any variable declared outside of its declarative region.

NOTE 6—A subprogram containing a wait statement shall not have a parent that is a subprogram declared within either a protected type declaration or a protected type body.

4.4 Subprogram instantiation declarations

A subprogram instantiation declaration defines an instance of an uninstantiated subprogram. The instance is called an *instantiated subprogram*.

subprogram_instantiation_declaration ::=
    subprogram_kind designator **is new** *uninstantiated_subprogram_*name [signature]
        [generic_map_aspect] ;

The uninstantiated subprogram name shall denote an uninstantiated subprogram declared in a subprogram declaration. The signature, if present, shall match the parameter and result type profile of exactly one subprogram denoted by the name, in which case the subprogram instantiation declaration defines an instance of the uninstantiated subprogram whose parameter and result type profile is matched by the signature. The subprogram kind shall repeat the reserved word used in the declaration of the uninstantiated subprogram. The generic map aspect, if present, optionally associates a single actual with each formal generic (or member thereof) in the corresponding subprogram declaration. Each formal generic (or member thereof) shall be associated at most once. The generic map aspect is described in 6.5.7.2.

The subprogram instantiation declaration is equivalent to a subprogram declaration and a subprogram body that jointly define a generic-mapped subprogram. The designator of the generic-mapped subprogram declaration and subprogram body is the designator of the subprogram instantiation declaration. The generic-mapped subprogram declaration and subprogram body have the generic list of the uninstantiated subprogram declaration, the generic map aspect of the subprogram instantiation declaration, and the parameter list and return type (if appropriate) of the uninstantiated subprogram declaration. The generic-mapped subprogram body has the declarations and statements of the uninstantiated subprogram body. The meaning of any identifier appearing anywhere in the generic-mapped subprogram declaration or subprogram body is that associated with the corresponding occurrence of the identifier in the subprogram instantiation declaration, the uninstantiated subprogram declaration, or the uninstantiated subprogram body, respectively, except that an identifier that denotes the uninstantiated subprogram denotes, instead, the generic-mapped subprogram.

If the subprogram instantiation declaration occurs immediately within an enclosing package declaration, the generic-mapped subprogram body occurs at the end of the package body corresponding to the enclosing package declaration. If there is no such body, then there is implicitly a package body corresponding to the enclosing package declaration, and that implicit body contains the generic-mapped subprogram body. If the subprogram instantiation declaration occurs immediately within an enclosing protected type declaration, the generic-mapped subprogram body occurs at the end of the protected type body corresponding to the enclosing protected type declaration.

NOTE—If two uninstantiated subprograms have the same name and have parameter and result type profiles that include formal generic types of the uninstantiated subprograms, in addition to other types, a signature can be used to distinguish between the uninstantiated subprograms, since the formal generic types are made visible by selection in the signatures.

4.5 Subprogram overloading

4.5.1 General

Two formal parameter lists are said to have the same *parameter type profile* if and only if they have the same number of parameters, and if at each parameter position the corresponding parameters have the same base type. Two subprograms are said to have the same *parameter and result type profile* if and only if both have the same parameter type profile, and if either both are functions with the same result base type or neither of the two is a function.

A given subprogram designator can be used to designate multiple subprograms. The subprogram designator is then said to be overloaded; the designated subprograms are also said to be overloaded and to overload each other. If two subprograms overload each other, one of them can hide the other only if both subprograms have the same parameter and result type profile.

A call to an overloaded subprogram is ambiguous (and therefore is an error) if the name of the subprogram, the number of parameter associations, the types and order of the actual parameters, the names of the formal parameters (if named associations are used), and the result type (for functions) are not sufficient to identify exactly one (overloaded) subprogram.

Similarly, a reference to an overloaded resolution function name in a subtype indication is ambiguous (and is therefore an error) if the name of the function, the number of formal parameters, the result type, and the relationships between the result type and the types of the formal parameters (as defined in 4.6) are not sufficient to identify exactly one (overloaded) subprogram specification.

Examples:

— Declarations of overloaded subprograms:

```
procedure Dump (F: inout Text; Value: Integer);
procedure Dump (F: inout Text; Value: String);

procedure Check (Setup: Time; signal D: Data; signal C: Clock);
procedure Check (Hold: Time; signal C: Clock; signal D: Data);
```

— Calls to overloaded subprograms:

```
Dump (Sys_Output, 12);
Dump (Sys_Error, "Actual output does not match expected output");

Check (Setup=>10 ns, D=>DataBus, C=>Clk1);
Check (Hold=>5 ns, D=>DataBus, C=>Clk2);
Check (15 ns, DataBus, Clk) ;
   --  Ambiguous if the base type of DataBus is the same type
   --  as the base type of Clk.
```

NOTE 1—The notion of parameter and result type profile does not include parameter names, parameter classes, parameter modes, parameter subtypes, or default expressions or their presence or absence.

NOTE 2—Ambiguities may (but need not) arise when actual parameters of the call of an overloaded subprogram are themselves overloaded function calls, literals, or aggregates. Ambiguities may also (but need not) arise when several overloaded subprograms belonging to different packages are visible. These ambiguities can usually be solved in two ways: qualified expressions can be used for some or all actual parameters and for the result, if any; or the name of the subprogram can be expressed more explicitly as an expanded name (see 8.3). Further, ambiguities may (but need not) arise when the declarations of overloaded subprograms in an uninstantiated declaration have parameter and result type profiles that involve different formal generic types of the uninstantiated declaration. If the declaration is instantiated with the same actual type associated with the formals, the resulting overloaded subprograms in the instance may have the same parameter and result type profile. Such ambiguities cannot be solved.

4.5.2 Operator overloading

The declaration of a function whose designator is an operator symbol is used to overload an operator. The sequence of characters of the operator symbol shall be one of the operators in the operator classes defined in 9.2.

The subprogram specification of a unary operator shall have a single parameter, unless the subprogram specification is a method (see 5.6.2) of a protected type. In this latter case, the subprogram specification shall have no parameters. The subprogram specification of a binary operator shall have two parameters, unless the subprogram specification is a method of a protected type, in which case, the subprogram specification shall have a single parameter. If the subprogram specification of a binary operator has two

parameters, for each use of this operator, the first parameter is associated with the left operand, and the second parameter is associated with the right operand.

For each of the operators "+", "–", "and", "or", "xor", "nand", "nor" and "xnor", overloading is allowed both as a unary operator and as a binary operator.

NOTE 1—Overloading of the equality operator does not affect the selection of choices in a case statement in a selected signal assignment statement, nor does it affect the propagation of signal values.

NOTE 2—A user-defined operator that has the same designator as a short-circuit operator (i.e., a user-defined operator that overloads the short-circuit operator) is not invoked in a short-circuit manner. Specifically, calls to the user-defined operator always evaluate both arguments prior to the execution of the function.

NOTE 3—Functions that overload operator symbols may also be called using function call notation rather than operator notation. This statement is also true of the predefined operators themselves.

Examples:

```
type MVL is ('0', '1', 'Z', 'X');
type MVL_Vector is array (Natural range <>) of MVL;
function "and" (Left, Right: MVL) return MVL;
function "or" (Left, Right: MVL) return MVL;
function "not" (Value: MVL) return MVL;
function "xor" (Right: MVL_Vector) return MVL;

signal Q,R,S,T: MVL;
signal V: MVL_Vector(0 to 3);

Q <= 'X' or '1';
R <= "or" ('0','Z');
S <= (Q and R) or not S;
T <= xor V;
```

4.5.3 Signatures

A signature distinguishes between overloaded subprograms and overloaded enumeration literals based on their parameter and result type profiles. A signature can be used in a subprogram instantiation declaration, attribute name, entity designator, or alias declaration.

signature ::= [[type_mark { , type_mark }] [**return** type_mark]]

(Note that the initial and terminal brackets are part of the syntax of signatures and do not indicate that the entire right-hand side of the production is optional.) A signature is said to match the parameter and the result type profile of a given subprogram if, and only if, all of the following conditions hold:

- The number of type marks prior to the reserved word **return**, if any, matches the number of formal parameters of the subprogram.
- At each parameter position, the base type denoted by the type mark of the signature is the same as the base type of the corresponding formal parameter of the subprogram.
- If the reserved word **return** is present, the subprogram is a function and the base type of the type mark following the reserved word in the signature is the same as the base type of the return type of the function, or the reserved word **return** is absent and the subprogram is a procedure.

Similarly, a signature is said to match the parameter and result type profile of a given enumeration literal if the signature matches the parameter and result type profile of the subprogram equivalent to the enumeration literal defined in 5.2.2.1.

Example:

```
attribute BuiltIn of "or" [MVL, MVL return MVL]: function is TRUE;
   --  Because of the presence of the signature, this attribute
   --  specification decorates only the "or" function defined in 4.5.2.

attribute Mapping of JMP [return OpCode] :literal is "001";
```

4.6 Resolution functions

A resolution function is a function that defines how the values of multiple sources of a given signal are to be resolved into a single value for that signal. Resolution functions are associated with signals that require resolution by including the name of the resolution function in the declaration of the signal or in the declaration of the subtype of the signal. A signal with an associated resolution function is called a resolved signal (see 6.4.2.3).

A resolution function shall be a pure function other than an uninstantiated function (see 4.2.1); moreover, it shall have a single input parameter of class **constant** that is a one-dimensional, unconstrained or partially constrained array with an undefined index range and whose element type is that of the associated subtype or subelement subtype in the subtype indication in which the name of the resolution function appears. The resolution function name shall not be an attribute name (see 8.6). The type of the return value of the function shall also be that of the associated subtype or subelement subtype in the subtype indication in which the name of the resolution function appears. Errors occur at the place of the subtype indication containing the name of the resolution function if any of these checks fail (see 6.3).

The resolution function associated with a resolved signal determines the *resolved value* of the signal as a function of the collection of inputs from its multiple sources. If a resolved signal is of a composite type, and if subelements of that type also have associated resolution functions, such resolution functions have no effect on the process of determining the resolved value of the signal. It is an error if a resolved signal has more connected sources than the number of elements in the index type of the unconstrained array type used to define the parameter of the corresponding resolution function.

Resolution functions are implicitly invoked during each simulation cycle in which corresponding resolved signals are active (see 14.7.3.1). Each time a resolution function is invoked, it is passed an array value, each element of which is determined by a corresponding source of the resolved signal, but excluding those sources that are drivers whose values are determined by null transactions (see 10.5.2.2). Such drivers are said to be *off*. For certain invocations (specifically, those involving the resolution of sources of a signal declared with the signal kind **bus**), a resolution function may thus be invoked with an input parameter that is a null array; this occurs when all sources of the bus are drivers, and they are all off. In such a case, the resolution function returns a value representing the value of the bus when no source is driving it.

Example:

```
function WIRED_OR (Inputs: BIT_VECTOR) return BIT is
   constant FloatValue: BIT := '0';
begin
   if Inputs'Length = 0 then
      --  This is a bus whose drivers are all off.
      return FloatValue;
   else
      for I in Inputs'Range loop
         if Inputs(I) = '1' then
            return '1';
```

```
        end if;
      end loop;
      return '0';
   end if;
end function WIRED_OR;
```

4.7 Package declarations

A package declaration defines the interface to a package. The scope of a declaration within a package can be extended to other design units or to other parts of the design unit containing the package declaration.

package_declaration ::=
    package identifier **is**
        package_header
        package_declarative_part
    end [**package**] [*package*_simple_name] ;

package_header ::=
    [generic_clause
    [generic_map_aspect ;]]

package_declarative_part ::=
    { package_declarative_item }

package_declarative_item ::=
      subprogram_declaration
    | subprogram_instantiation_declaration
    | package_declaration
    | package_instantiation_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | component_declaration
    | attribute_declaration
    | attribute_specification
    | disconnection_specification
    | use_clause
    | group_template_declaration
    | group_declaration
    | *PSL*_Property_Declaration
    | *PSL*_Sequence_Declaration

If a simple name appears at the end of the package declaration, it shall repeat the identifier of the package declaration.

If the package header is empty, the package declared by a package declaration is called a *simple package*. If the package header contains a generic clause and no generic map aspect, the package is called an *uninstantiated package*. If the package header contains both a generic clause and a generic map aspect, the package is called a *generic-mapped package*. A package declared with a generic clause in which every

generic declaration has a default, and with no generic map aspect, is considered to be an uninstantiated package, not a generic-mapped package with default associations for all of the generic declarations.

If a package declarative item is a full type declaration whose type definition is a protected type definition, then that protected type definition shall not be a protected type body.

Items declared immediately within a simple or a generic-mapped package declaration become visible by selection within a given design unit wherever the name of that package is visible in the given unit. Such items may also be made directly visible by an appropriate use clause (see 12.4). Items declared immediately within an uninstantiated package declaration cannot be made visible outside of the package.

For a package declaration that appears in a subprogram body, a process statement, or a protected type body, it is an error if a variable declaration in the package declarative part of the package declaration declares a shared variable. Moreover, it is an error if a signal declaration, a disconnection specification, or a PSL declaration appears as a package declarative item of such a package declaration.

NOTE—Not all packages will have a package body. In particular, a package body is unnecessary if no subprograms, deferred constants, or protected type definitions are declared in the package declaration.

Examples:

— A package declaration that needs no package body:

```
package TimeConstants is
   constant tPLH: Time := 10 ns;
   constant tPHL: Time := 12 ns;
   constant tPLZ: Time := 7 ns;
   constant tPZL: Time := 8 ns;
   constant tPHZ: Time := 8 ns;
   constant tPZH: Time := 9 ns;
end TimeConstants;
```

— A package declaration that needs a package body:

```
package TriState is
   type Tri is ('0', '1', 'Z', 'E');
   function BitVal (Value: Tri) return Bit;
   function TriVal (Value: Bit) return Tri;
   type TriVector is array (Natural range <>) of Tri;
   function Resolve (Sources: TriVector) return Tri;
end package TriState;
```

4.8 Package bodies

A package body defines the bodies of subprograms and the values of deferred constants declared in the interface to the package.

package_body ::=
    package body *package*_simple_name **is**
        package_body_declarative_part
    end [**package body**] [*package*_simple_name] ;

package_body_declarative_part ::=
    { package_body_declarative_item }

package_body_declarative_item ::=
      subprogram_declaration

| subprogram_body
| subprogram_instantiation_declaration
| package_declaration
| package_body
| package_instantiation_declaration
| type_declaration
| subtype_declaration
| constant_declaration
| variable_declaration
| file_declaration
| alias_declaration
| attribute_declaration
| attribute_specification
| use_clause
| group_template_declaration
| group_declaration

The simple name at the start of a package body shall repeat the package identifier. If a simple name appears at the end of the package body, it shall be the same as the identifier in the package declaration.

A package body that is not a library unit shall appear immediately within the same declarative region as the corresponding package declaration and textually subsequent to that package declaration.

For a package body that appears in a subprogram body, a process statement or a protected type body, it is an error if a variable declaration in the package body declarative part of the package body declares a shared variable.

In addition to subprogram body and constant declarative items, a package body may contain certain other declarative items to facilitate the definition of the bodies of subprograms declared in the interface. Items declared in the body of a package cannot be made visible outside of the package body.

If a given package declaration contains a deferred constant declaration (see 6.4.2.2), then a constant declaration with the same identifier shall appear as a declarative item in the corresponding package body. This object declaration is called the *full* declaration of the deferred constant. The subtype indication given in the full declaration shall lexically conform to that given in the deferred constant declaration.

Within a package declaration that contains the declaration of a deferred constant, and within the body of that package (before the end of the corresponding full declaration), the use of a name that denotes the deferred constant is only allowed in the default expression for a local generic, local port, or formal parameter. The result of evaluating an expression that references a deferred constant before the elaboration of the corresponding full declaration is not defined by the language.

Example:

```
package body TriState is

   function BitVal (Value: Tri) return Bit is
      constant Bits : Bit_Vector := "0100";
   begin
      return Bits(Tri'Pos(Value));
   end;

   function TriVal (Value: Bit) return Tri is
   begin
```



```
      return Tri'Val(Bit'Pos(Value));
   end;

   function Resolve (Sources: TriVector) return Tri is
      variable V: Tri := 'Z';
   begin
      for i in Sources'Range loop
         if Sources(i) /= 'Z' then
            if V = 'Z' then
               V := Sources(i);
            else
               return 'E';
            end if;
         end if;
      end loop;
      return V;
   end;

end package body TriState;
```

4.9 Package instantiation declarations

A package instantiation declaration defines an instance of an uninstantiated package. The instance is called an *instantiated package*.

package_instantiation_declaration ::=
    package identifier **is new** *uninstantiated_package_*name
        [generic_map_aspect] ;

The uninstantiated package name shall denote an uninstantiated package declared in a package declaration. The generic map aspect, if present, optionally associates a single actual with each formal generic (or member thereof) in the corresponding package declaration. Each formal generic (or member thereof) shall be associated at most once. The generic map aspect is described in 6.5.7.2.

The package instantiation declaration is equivalent to declaration of a generic-mapped package, consisting of a package declaration and possibly a corresponding package body. The simple name of the generic-mapped package declaration is the identifier of the package instantiation declaration. The generic-mapped package declaration has the generic clause of the uninstantiated package declaration, the generic map aspect of the package instantiation declaration, and the declarations of the uninstantiated package declaration. The package body corresponding to the generic-mapped package is present if the uninstantiated package has a package body. In that case, the simple name of the generic-mapped package body is the identifier of the package instantiation declaration, and the declarations of the generic-mapped package body are the declarations of the uninstantiated package body. The meaning of any identifier appearing anywhere in the generic-mapped package declaration or package body is that associated with the corresponding occurrence of the identifier in the package instantiation declaration, the uninstantiated package declaration, or the uninstantiated package body, respectively, except that an identifier that denotes the uninstantiated package denotes, instead, the generic-mapped package.

If the package instantiation declaration occurs immediately within an enclosing package declaration and the uninstantiated package has a package body, the generic-mapped package body occurs at the end of the package body corresponding to the enclosing package declaration. If there is no such body, then there is implicitly a package body corresponding to the enclosing package declaration, and that implicit body contains the generic-mapped package body.

4.10 Conformance rules

Whenever the language rules either require or allow the specification of a given subprogram to be provided in more than one place, the following variations are allowed at each place:

— A numeric literal can be replaced by a different numeric literal if and only if both have the same value.

— A simple name can be replaced by an expanded name in which this simple name is the suffix if, and only if, at both places the meaning of the simple name is given by the same declaration.

Two subprogram specifications are said to *lexically conform* if, apart from comments and the preceding allowed variations, both specifications are formed by the same sequence of lexical elements and if corresponding lexical elements are given the same meaning by the visibility rules.

Lexical conformance is likewise defined for subtype indications in deferred constant declarations.

Two subprogram declarations are said to have *conforming profiles* if and only if both are procedures or both are functions, the parameter and result type profiles of the subprograms are the same and, at each parameter position, the corresponding parameters have the same class and mode.

NOTE 1—A simple name can be replaced by an expanded name even if the simple name is itself the prefix of a selected name. For example, Q.R can be replaced by P.Q.R if Q is declared immediately within P.

NOTE 2—The subprogram specification of an impure function is never lexically conformant to a subprogram specification of a pure function.

NOTE 3—The following specifications do not lexically conform since they are not formed by the same sequence of lexical elements:

```
procedure P (X,Y: INTEGER)
procedure P (X: INTEGER; Y: INTEGER)
procedure P (X,Y: in INTEGER)
```

NOTE 4—Conformance of profiles is required for formal and actual generic subprograms (see 6.5.4).

5. Types

5.1 General

This clause describes the various categories of types that are provided by the language as well as those specific types that are predefined. The declarations of all predefined types are contained in package STANDARD, the declaration of which appears in Clause 16.

A type is characterized by a set of values and a set of operations. The set of operations of a type includes the explicitly declared subprograms that have a parameter or result of the type. The remaining operations of a type are the basic operations and the predefined operations (see 5.2.6, 5.3.2.4, 5.4.3, and 5.5.2). These operations are each implicitly declared for a given type declaration immediately after the type declaration and before the next explicit declaration, if any.

A *basic operation* is an operation that is inherent in one of the following:

- An assignment (in assignment statements and initializations)
- An allocator
- A selected name, an indexed name, or a slice name
- A qualification (in a qualified expression), an explicit type conversion, a formal or actual part in the form of a type conversion, or an implicit type conversion of a value of type *universal_integer* or *universal_real* to the corresponding value of another numeric type
- A numeric literal (for a universal type), the literal **null** (for an access type), a string literal, a bit string literal, an aggregate, or a predefined attribute

There are five classes of types. *Scalar* types are integer types, floating-point types, physical types, and types defined by an enumeration of their values; values of these types have no elements. *Composite* types are array and record types; values of these types consist of element values. *Access* types provide access to objects of a given type. *File* types provide access to objects that contain a sequence of values of a given type. *Protected types* provide atomic and exclusive access to variables accessible to multiple processes.

The set of possible values for an object of a given type can be subjected to a condition that is called a *constraint* (the case where the constraint imposes no restriction is also included); a value is said to satisfy a constraint if it satisfies the corresponding condition. A *subtype* is a type together with a constraint. A value is said to *belong to a subtype* of a given type if it belongs to the type and satisfies the constraint; the given type is called the *base type* of the subtype. A type is a subtype of itself; such a subtype is said to be *unconstrained* (it corresponds to a condition that imposes no restriction). The base type of a type is the type itself.

A composite subtype is said to be *unconstrained* if:

- It is an array subtype with no index constraint and the element subtype either is not a composite subtype or is an unconstrained composite type, or
- It is a record subtype with at least one element of a composite subtype and each element that is of a composite subtype is unconstrained.

A composite subtype is said to be *fully constrained* if:

- It is an array subtype with an index constraint and the element subtype either is not a composite subtype or is a fully constrained composite type, or
- It is a record subtype and each element subtype either is not a composite subtype or is a fully constrained composite subtype.

A composite subtype is said to be *partially constrained* if it is neither unconstrained nor fully constrained.

The set of operations defined for a subtype of a given type includes the operations defined for the type; however, the assignment operation to an object having a given subtype only assigns values that belong to the subtype. Additional operations, such as qualification (in a qualified expression) are implicitly defined by a subtype declaration.

The term *subelement* is used in this standard in place of the term *element* to indicate either an element, or an element of another element or subelement. Where other subelements are excluded, the term *element* is used instead.

A given type shall not have a subelement whose type is the given type itself.

A *member* of an object is one of the following:

— A slice of the object
— A subelement of the object
— A slice of a subelement of the object

The name of a class of types is used in this standard as a qualifier for objects and values that have a type of the class considered. For example, the term *array object* is used for an object whose type is an array type; similarly, the term *access value* is used for a value of an access type.

NOTE 1—The set of values of a subtype is a subset of the values of the base type. This subset need not be a proper subset.

NOTE 2—All composite subelements of an unconstrained type are unconstrained.

5.2 Scalar types

5.2.1 General

Scalar types consist of *enumeration types, integer types, physical types*, and *floating-point types*. Enumeration types and integer types are called *discrete* types. Integer types, floating-point types, and physical types are called *numeric* types. All scalar types are ordered; that is, all relational operators are predefined for their values. Each value of a discrete or physical type has a position number that is an integer value.

```
scalar_type_definition ::=
      enumeration_type_definition
    | integer_type_definition
    | floating_type_definition
    | physical_type_definition

range_constraint ::= range range

range ::=
      range_attribute_name
    | simple_expression direction simple_expression

direction ::= to | downto
```

A range specifies a subset of values of a scalar type. A range is said to be a *null* range if the specified subset is empty.

The range L **to** R is called an *ascending* range; if L > R, then the range is a null range. The range L **downto** R is called a *descending range*; if L < R, then the range is a null range. L is called the *left bound* of the range,

and R is called the *right bound* of the range. The *lower bound* of a range is the left bound if the range is ascending or the right bound if the range is descending. The *upper bound* of a range is the right bound if the range is ascending or the left bound if the range is descending. The value V is said to *belong to the range* if the relations (*lower bound* <= V) and (V <= *upper bound*) are both true. The operators >, <, and <= in the preceding definitions are the predefined operators of the applicable scalar type.

For values of discrete or physical types, a value V1 is said to be *to the left* of a value V2 within a given range if both V1 and V2 belong to the range and either the range is an ascending range and V2 is the successor of V1, or the range is a descending range and V2 is the predecessor of V1. A list of values of a given range is in *left to right order* if each value in the list is to the left of the next value in the list within that range, except for the last value in the list.

If a range constraint is used in a subtype indication, the type of the expressions (likewise, of the bounds of a range attribute) shall be the same as the base type of the type mark of the subtype indication. A range constraint is *compatible* with a subtype if each bound of the range belongs to the subtype or if the range constraint defines a null range. Otherwise, the range constraint is not compatible with the subtype.

A subtype S1 is *compatible* with a subtype S2 if the range constraint associated with S1 is compatible with S2.

The direction of a range constraint is the same as the direction of its range.

NOTE—Indexing and iteration rules use values of discrete types.

5.2.2 Enumeration types

5.2.2.1 General

An enumeration type definition defines an enumeration type.

```
enumeration_type_definition ::=
    ( enumeration_literal { , enumeration_literal } )

enumeration_literal ::=  identifier | character_literal
```

The identifiers and character literals listed by an enumeration type definition shall be distinct within the enumeration type definition. Each enumeration literal is the declaration of the corresponding enumeration literal. For the purpose of determining the parameter and result type profile of an enumeration literal, this declaration is equivalent to the declaration of a parameterless function whose designator is the same as the enumeration literal and whose result type is the same as the enumeration type; the declaration is, nonetheless, a declaration of a literal, not of a function.

An enumeration type is said to be a *character type* if at least one of its enumeration literals is a character literal.

Each enumeration literal yields a different enumeration value. The predefined order relations between enumeration values follow the order of corresponding position numbers. The position number of the value of the first listed enumeration literal is zero; the position number for each additional enumeration literal is one more than that of its predecessor in the list.

If the same identifier or character literal is specified in more than one enumeration type definition, the corresponding literals are said to be *overloaded*. At any place where an overloaded enumeration literal occurs in the text of a program, the type of the enumeration literal is determined according to the rules for overloaded subprograms (see 4.5).

Each enumeration type definition defines an ascending range.

Examples:

```
type MULTI_LEVEL_LOGIC is (LOW, HIGH, RISING, FALLING, AMBIGUOUS);

type BIT is ('0','1');

type SWITCH_LEVEL is ('0','1','X');     --  Overloads '0' and '1'
```

5.2.2.2 Predefined enumeration types

The predefined enumeration types are CHARACTER, BIT, BOOLEAN, SEVERITY_LEVEL, FILE_OPEN_KIND, and FILE_OPEN_STATUS.

The predefined type CHARACTER is a character type whose values are the 256 characters of the ISO/IEC 8859-1 character set. Each of the 191 graphic characters of this character set is denoted by the corresponding character literal.

The declarations of the predefined types CHARACTER, BIT, BOOLEAN, SEVERITY_LEVEL, FILE_OPEN_KIND, and FILE_OPEN_STATUS appear in package STANDARD in Clause 16.

NOTE 1—The first 33 nongraphic elements of the predefined type CHARACTER (from NUL through DEL) are the ASCII abbreviations for the nonprinting characters in the ASCII set (except for those noted in Clause 16). The ASCII names are chosen as ISO/IEC 8859-1:1998 does not assign them abbreviations. The next 32 (C128 through C159) are also not assigned abbreviations, so names unique to VHDL are assigned.

NOTE 2—Type BOOLEAN can be used to model either active high or active low logic depending on the particular conversion functions chosen to and from type BIT.

5.2.3 Integer types

5.2.3.1 General

An integer type definition defines an integer type whose set of values includes those of the specified range.

integer_type_definition ::= range_constraint

An integer type definition defines both a type and a subtype of that type. The type is an anonymous type, the range of which is selected by the implementation; this range shall be such that it wholly contains the range given in the integer type definition. The subtype is a named subtype of this anonymous base type, where the name of the subtype is that given by the corresponding type declaration and the range of the subtype is the given range.

Each bound of a range constraint that is used in an integer type definition shall be a locally static expression of some integer type, but the two bounds need not have the same integer type. (Negative bounds are allowed.)

Integer literals are the literals of an anonymous predefined type that is called *universal_integer* in this standard. Other integer types have no literals. However, for each integer type there exists an implicit conversion that converts a value of type *universal_integer* into the corresponding value (if any) of the integer type (see 9.3.6).

The position number of an integer value is the corresponding value of the type *universal_integer*.

The same arithmetic operators are predefined for all integer types (see 9.2). It is an error if the execution of such an operation (in particular, an implicit conversion) cannot deliver the correct result (that is, if the value corresponding to the mathematical result is not a value of the integer type).

An implementation may restrict the bounds of the range constraint of integer types other than type *universal_integer*. However, an implementation shall allow the declaration of any integer type whose range is wholly contained within the bounds –2147483647 and +2147483647 inclusive.

Examples:

```
type TWOS_COMPLEMENT_INTEGER is range -32768 to 32767;

type BYTE_LENGTH_INTEGER is range 0 to 255;

type WORD_INDEX is range 31 downto 0;

subtype HIGH_BIT_LOW is BYTE_LENGTH_INTEGER range 0 to 127;
```

5.2.3.2 Predefined integer types

The only predefined integer type is the type INTEGER. The range of INTEGER is implementation dependent, but it is guaranteed to include the range –2147483647 to +2147483647. It is defined with an ascending range.

NOTE—The range of INTEGER in a particular implementation is determinable from the values of its 'LOW and 'HIGH attributes.

5.2.4 Physical types

5.2.4.1 General

Values of a physical type represent measurements of some quantity. Any value of a physical type is an integral multiple of the primary unit of measurement for that type.

physical_type_definition ::=
    range_constraint
        units
            primary_unit_declaration
            { secondary_unit_declaration }
        end units [*physical_type*_simple_name]

primary_unit_declaration ::= identifier ;

secondary_unit_declaration ::= identifier = physical_literal ;

physical_literal ::= [abstract_literal] *unit*_name

A physical type definition defines both a type and a subtype of that type. The type is an anonymous type, the range of which is selected by the implementation; this range shall be such that it wholly contains the range given in the physical type definition. The subtype is a named subtype of this anonymous base type, where the name of the subtype is that given by the corresponding type declaration and the range of the subtype is the given range.

Each bound of a range constraint that is used in a physical type definition shall be a locally static expression of some integer type, but the two bounds need not have the same integer type. (Negative bounds are allowed.)

Each unit declaration (either the primary unit declaration or a secondary unit declaration) defines a *unit name*. Unit names declared in secondary unit declarations shall be directly or indirectly defined in terms of integral multiples of the primary unit of the type declaration in which they appear. The position numbers of unit names need not lie within the range specified by the range constraint.

If a simple name appears at the end of a physical type declaration, it shall repeat the identifier of the type declaration in which the physical type definition is included.

The abstract literal portion (if present) of a physical literal appearing in a secondary unit declaration shall be an integer literal.

A physical literal consisting solely of a unit name is equivalent to the integer 1 followed by the unit name.

There is a position number corresponding to each value of a physical type. The position number of the value corresponding to a unit name is the number of primary units represented by that unit name. The position number of the value corresponding to a physical literal with an abstract literal part is the largest integer that is not greater than the product of the value of the abstract literal and the position number of the accompanying unit name.

The same arithmetic operators are predefined for all physical types (see 9.2). It is an error if the execution of such an operation cannot deliver the correct result (i.e., if the value corresponding to the mathematical result is not a value of the physical type).

An implementation may restrict the bounds of the range constraint of a physical type. However, an implementation shall allow the declaration of any physical type whose range is wholly contained within the bounds –2147483647 and +2147483647 inclusive.

Examples:

```
type DURATION is range -1E18 to 1E18
      units
         fs;                          --femtosecond
         ps    =   1000 fs;           --picosecond
         ns    =   1000 ps;           --nanosecond
         us    =   1000 ns;           --microsecond
         ms    =   1000 us;           --millisecond
         sec   =   1000 ms;           --second
         min   =   60 sec;            --minute
      end units;

type DISTANCE is range 0 to 1E16
      units
      -- primary unit:
         Å;                           --angstrom

      -- metric lengths:
         nm    =   10 Å;              --nanometer
         um    =   1000 nm;           --micrometer (or micron)
         mm    =   1000 um;           --millimeter
         cm    =   10 mm;             --centimeter
```



```
        m     =   1000 mm;          --meter
        km    =   1000 m;           --kilometer

    -- English lengths:
        mil   =   254000 Å;         --mil
        inch  =   1000 mil;         --inch
        ft    =   12 inch;          --foot
        yd    =   3 ft;             --yard
        fm    =   6 ft;             --fathom
        mi    =   5280 ft;          --mile
        lg    =   3 mi;             --league
    end units DISTANCE;

variable x: distance;
variable y: duration;
variable z: integer;

x := 5 Å + 13 ft - 27 inch;
y := 3 ns + 5 min;
z := ns / ps;
x := z * mi;
y := y/10;
z := 39.34 inch / m;
```

NOTE 1—The 'POS and 'VAL attributes may be used to convert between abstract values and physical values.

NOTE 2—The value of a physical literal, whose abstract literal is either the integer value zero or the floating-point value zero, is the same value (specifically zero primary units) no matter what unit name follows the abstract literal.

5.2.4.2 Predefined physical types

The only predefined physical type is type TIME. The range of TIME is implementation dependent, but it is guaranteed to include the range –2147483647 to +2147483647. It is defined with an ascending range. All specifications of delays and pulse rejection limits shall be of type TIME. The declaration of type TIME appears in package STANDARD in Clause 16.

By default, the primary unit of type TIME (1 fs) is the *resolution limit* for type TIME. Any TIME value whose absolute value is smaller than this limit is truncated to zero (0) time units. An implementation may allow a given elaboration of a model (see Clause 14) to select a secondary unit of type TIME as the resolution limit. Furthermore, an implementation may restrict the precision of the representation of values of type TIME and the results of expressions of type TIME, provided that values as small as the resolution limit are representable within those restrictions. It is an error if a given unit of type TIME appears anywhere within the design hierarchy defining a model to be elaborated, and if the position number of that unit is less than that of the secondary unit selected as the resolution limit for type TIME during the elaboration of the model, unless that unit is part of a physical literal whose abstract literal is either the integer value zero or the floating-point value zero.

NOTE—By selecting a secondary unit of type TIME as the resolution limit for type TIME, it may be possible to simulate for a longer period of simulated time, with reduced accuracy, or to simulate with greater accuracy for a shorter period of simulated time.

Cross-references: Delay and rejection limit in a signal assignment, 10.5; disconnection, delay of a guarded signal, 7.4; function NOW, 16.3; predefined attributes, functions of TIME, 16.2; simulation time, 14.7.3 and 14.7.4; type TIME, 16.3; updating a projected waveform, 10.5.2.2; wait statements, timeout clause in, 10.2; elaboration of a declarative part, 14.4.

5.2.5 Floating-point types

5.2.5.1 General

Floating-point types provide approximations to the real numbers.

floating_type_definition ::= range_constraint

A floating type definition defines both a type and a subtype of that type. The type is an anonymous type, the range of which is selected by the implementation; this range shall be such that it wholly contains the range given in the floating type definition. The subtype is a named subtype of this anonymous base type, where the name of the subtype is that given by the corresponding type declaration and the range of the subtype is the given range.

Each bound of a range constraint that is used in a floating type definition shall be a locally static expression of some floating-point type, but the two bounds need not have the same floating-point type. (Negative bounds are allowed.)

Floating-point literals are the literals of an anonymous predefined type that is called *universal_real* in this standard. Other floating-point types have no literals. However, for each floating-point type there exists an implicit conversion that converts a value of type *universal_real* into the corresponding value (if any) of the floating-point type (see 9.3.6).

The same arithmetic operations are predefined for all floating-point types (see 9.2). A design is erroneous if the execution of such an operation cannot deliver the correct result (that is, if the value corresponding to the mathematical result is not a value of the floating-point type).

An implementation shall choose a representation for all floating-point types except for *universal_real* that conforms either to IEEE Std 754-1985 or to IEEE Std 854-1987; in either case, a minimum representation size of 64 bits is required for this *chosen representation*.

An implementation may restrict the bounds of the range constraint of floating-point types other than type *universal_real*. However, an implementation shall allow the declaration of any floating-point type whose range is wholly contained within the bounds allowed by the chosen representation.

NOTE—An implementation is not required to detect errors in the execution of a predefined floating-point arithmetic operation, since the detection of overflow conditions resulting from such operations might not be easily accomplished on many host systems.

5.2.5.2 Predefined floating-point types

The only predefined floating-point type is the type REAL. The range of REAL is host-dependent, but it is guaranteed to be the largest allowed by the chosen representation. It is defined with an ascending range.

NOTE—The range of REAL in a particular implementation is determinable from the values of its 'LOW and 'HIGH attributes.

5.2.6 Predefined operations on scalar types

Given a type declaration that declares a scalar type T, the following operations are implicitly declared immediately following the type declaration (except for the TO_STRING operations in package STANDARD, which are implicitly declared at the end of the package declaration):

```
function MINIMUM (L, R: T) return T;
function MAXIMUM (L, R: T) return T;
```



```
function TO_STRING (VALUE: T) return STRING;
```

The MINIMUM operation returns the value of L if L < R, or the value of R otherwise. The MAXIMUM operation returns the value of R if L < R, or the value of L otherwise. For both operations, the comparison is performed using the predefined relational operator for the type.

The TO_STRING operation returns the string representation (see 5.7) of the value of its actual parameter. The result type of the operation is the type STRING defined in package STANDARD.

The following operations are implicitly declared in package STD.STANDARD immediately following the declaration of the type BOOLEAN:

```
function RISING_EDGE  (signal S: BOOLEAN) return BOOLEAN;
function FALLING_EDGE (signal S: BOOLEAN) return BOOLEAN;
```

The function RISING_EDGE applied to a signal S of type BOOLEAN is TRUE if the expression "S'EVENT **and** S" is TRUE, and FALSE otherwise. The function FALLING_EDGE applied to a signal S of type BOOLEAN is TRUE if the expression "S'EVENT **and not** S" is TRUE, and FALSE otherwise.

The following operations are implicitly declared in package STD.STANDARD immediately following the declaration of the type BIT:

```
function RISING_EDGE  (signal S: BIT) return BOOLEAN;
function FALLING_EDGE (signal S: BIT) return BOOLEAN;
```

The function RISING_EDGE applied to a signal S of type BIT is TRUE if the expression "S'EVENT **and** S = '1'" is TRUE, and FALSE otherwise. The function FALLING_EDGE applied to a signal S of type BIT is TRUE if the expression "S'EVENT **and** S = '0'" is TRUE, and FALSE otherwise.

The following operation is implicitly declared in package STD.STANDARD at the end of the package declaration:

```
function TO_STRING (VALUE: TIME; UNIT: TIME) return STRING;
```

This overloaded TO_STRING operation returns the string representation (see 5.7) of the value of its actual parameter. The result type of the operation is the type STRING defined in package STANDARD. The parameter UNIT specifies how the result is to be formatted. The value of this parameter shall be equal to one of the units declared as part of the declaration of type TIME; the result is that the TIME value is formatted as an integer or real literal representing the number of multiples of this unit, followed by the name of the unit itself.

The following operations are implicitly declared in package STD.STANDARD at the end of the package declaration:

```
function TO_STRING (VALUE: REAL; DIGITS: NATURAL) return STRING;
function TO_STRING (VALUE: REAL; FORMAT: STRING) return STRING;
```

These overloaded TO_STRING operations return the value of the VALUE parameter converted to a string whose format is specified by the value of the DIGITS or FORMAT parameter, respectively. The result type of the operations is the type STRING defined in package STANDARD.

For the operation with the DIGITS parameter, the result is the string representation of the value. The DIGITS specifies how many digits appear to the right of the decimal point. If DIGITS is 0, then the string

representation is the same as that produced by the TO_STRING operation without the DIGITS or FORMAT parameter. If DIGITS is non-zero, then the string representation contains an integer part followed by '.' followed by the fractional part, using the specified number of digits, and no exponent (e.g., 3.14159).

For the operation with the FORMAT parameter, the format of the result is determined using the value of the FORMAT parameter in the manner described in ISO/IEC 8859-1:1998, ISO/IEC 9899:1999/Cor 1:2001, and ISO/IEC 9899:1999/Cor 2:2004 for the C `fprintf` function. A model is erroneous if it calls the operation with a value for the FORMAT parameter that is other than a conversion specification in which the conversion specifier is one of e, E, f, F, g, G, a, or A. Moreover, the model is erroneous if the conversion specification contains a length modifier or uses an asterisk for the field width or precision. An implementation shall support use of the conversion specifiers e, E, f, g, and G, and may additionally support use of the conversion specifiers F, a, and A. A model is erroneous if it calls the operation with a value for the FORMAT parameter that is a conversion specification in which the conversion specifier is one of F, a, or A and the implementation does not support use of the conversion specifier. The values of FLT_RADIX and DECIMAL_DIG (described in ISO/IEC 8859-1:1998, ISO/IEC 9899:1999/Cor 1:2001, and ISO/IEC 9899:1999/Cor 2:2004) are implementation defined.

5.3 Composite types

5.3.1 General

Composite types are used to define collections of values. These include both arrays of values (collections of values of a homogeneous type) and records of values (collections of values of potentially heterogeneous types).

composite_type_definition ::=
    array_type_definition
  | record_type_definition

An object of a composite type represents a collection of objects, one for each element of the composite object. It is an error if a composite type contains elements of file types or protected types. Thus an object of a composite type ultimately represents a collection of objects of scalar or access types, one for each noncomposite subelement of the composite object.

5.3.2 Array types

5.3.2.1 General

An array object is a composite object consisting of elements that have the same subtype. The name for an element of an array uses one or more index values belonging to specified discrete types. The value of an array object is a composite value consisting of the values of its elements.

array_type_definition ::=
  unbounded_array_definition | constrained_array_definition

unbounded_array_definition ::=
  array (index_subtype_definition { , index_subtype_definition })
    of *element*_subtype_indication

constrained_array_definition ::=
  array index_constraint **of** *element*_subtype_indication

index_subtype_definition ::= type_mark **range** <>

array_constraint ::=
    index_constraint [array_element_constraint]
  | (**open**) [array_element_constraint]

array_element_constraint ::= element_constraint

index_constraint ::= (discrete_range { , discrete_range })

discrete_range ::= *discrete*_subtype_indication | range

An array constraint may be used to constrain an array type or subtype (see 5.3.2.2 and 6.3).

An array object is characterized by the number of indices (the dimensionality of the array); the type, position, and range of each index; and the type and possible constraints of the elements. The order of the indices is significant.

A one-dimensional array has a distinct element for each possible index value. A multidimensional array has a distinct element for each possible sequence of index values that can be formed by selecting one value for each index (in the given order). The possible values for a given index are all the values that belong to the corresponding range; this range of values is called the *index range*.

An unbounded array definition in which the element subtype indication denotes either an unconstrained composite subtype or a subtype that is not a composite subtype defines an array type and a name denoting that type. For each object that has the array type, the number of indices, the type and position of each index, and the subtype of the elements are as in the type definition. The *index subtype* for a given index position is, by definition, the subtype denoted by the type mark of the corresponding index subtype definition. The values of the left and right bounds of each index range are not defined, but shall belong to the corresponding index subtype; similarly, the direction of each index range is not defined. The symbol <> (called a *box*) in an index subtype definition stands for an undefined range (different objects of the type need not have the same bounds and direction).

An unbounded array definition in which the element subtype indication denotes a partially or fully constrained composite subtype defines both an array type and a subtype of this type:

— The array type is an implicitly declared anonymous type; this type is defined by an implicit unbounded array definition, in which the element subtype indication denotes the base type of the subtype denoted by the element subtype indication of the explicit unbounded array definition and in which the index subtype definitions are those of the explicit unbounded array definition, in the same order.

— The array subtype is the subtype obtained by imposition of the constraint of the subtype denoted by the element subtype indication of the explicit unbounded array definition as an array element constraint on the array type.

A constrained array definition similarly defines both an array type and a subtype of this type:

— The array type is an implicitly declared anonymous type; this type is defined by an (implicit) unbounded array definition, in which the element subtype indication either denotes the base type of the subtype denoted by the element subtype indication of the constrained array definition, if that subtype is a composite type, or otherwise is the element subtype indication of the constrained array definition, in which the type mark of each index subtype definition denotes the subtype defined by the corresponding discrete range.

— The array subtype is the subtype obtained by imposition of the index constraint on the array type and, if the element subtype indication of the constrained array definition denotes a fully or partially

constrained composite subtype, imposition of the constraint of that subtype as an array element constraint on the array type.

If an array definition that defines both an array type and a subtype of that type is given for a type declaration, the simple name declared by this declaration denotes the array subtype.

The direction of a discrete range is the same as the direction of the range or the discrete subtype indication that defines the discrete range. If a subtype indication appears as a discrete range, the subtype indication shall not contain a resolution indication.

Examples:

— Examples of fully constrained array declarations:

```
type MY_WORD is array (0 to 31) of BIT;
   --  A memory word type with an ascending range.

type DATA_IN is array (7 downto 0) of FIVE_LEVEL_LOGIC;
   --  An input port type with a descending range.
```

— Example of partially constrained array declarations:

```
type MEMORY is array (INTEGER range <>) of MY_WORD;
   --  A memory array type.
```

— Example of unconstrained array declarations:

```
type SIGNED_FXPT is array (INTEGER range <>) of BIT;
   -- A signed fixed-point array type

type SIGNED_FXPT_VECTOR is array (NATURAL range <>) of SIGNED_FXPT;
   -- A vector of signed fixed-point elements
```

— Example of partially constrained array declarations:

```
type SIGNED_FXPT_5x4 is array (1 to 5, 1 to 4) of SIGNED_FXPT;
   -- A matrix of signed fixed-point elements
```

— Examples of array object declarations:

```
signal DATA_LINE: DATA_IN;
   --  Defines a data input line.

variable MY_MEMORY: MEMORY (0 to 2**n-1);
   --  Defines a memory of 2^n 32-bit words.

signal FXPT_VAL: SIGNED_FXPT (3 downto -4);
   -- Defines an 8-bit fixed-point signal

signal VEC: SIGNED_FXPT_VECTOR (1 to 20)(9 downto 0);
   -- Defines a vector of 20 10-bit fixed-point elements

variable SMATRIX: SIGNED_FXPT_5x4 (open)(3 downto -4);
   -- Defines a 5x4 matrix of 8-bit fixed-point elements
```

NOTE—The rules concerning constrained type declarations mean that a type declaration with a constrained array definition such as

```
type T is array (POSITIVE range MIN_BOUND to MAX_BOUND) of ELEMENT;
```

is equivalent to the sequence of declarations


```
subtype index_subtype is POSITIVE range MIN_BOUND to MAX_BOUND;
type array_type is array (index_subtype range <>) of ELEMENT'BASE;
subtype T is array_type (index_subtype)element_constraint;
```

where *index_subtype* and *array_type* are both anonymous and *element_constraint* is the constraint that applies to the subtype ELEMENT. Consequently, T is the name of a subtype and all objects declared with this type mark are arrays that have the same index range.

Similarly, a type declaration with an unbounded array definition whose element subtype indication denotes a partially or fully constrained subtype such as

```
type T is array (INTEGER range <>) of STRING(1 to 10);
```

is equivalent to the sequence of declarations

```
type array_type is array (INTEGER range <>) of STRING'BASE;
subtype T is array_type (open)(1 to 10);
```

5.3.2.2 Index constraints and discrete ranges

An index constraint determines the index range for every index of an array type and, thereby, the corresponding array bounds.

For a discrete range used in a constrained array definition and defined by a range, an implicit conversion to the predefined type INTEGER is assumed if the type of both bounds (prior to the implicit conversion) is the type *universal_integer*. Otherwise, the type of the range shall be determined by applying the rules of 12.5 to the range, considered as a complete context, using the rules that the type shall be discrete and that both bounds shall have the same type. These rules apply also to a discrete range used in a loop parameter specification (see 10.10) or a generate parameter specification (see 11.8).

If an array constraint of the first form (including an index constraint) applies to a type or subtype, then the type or subtype shall be an unconstrained or partially constrained array type with no index constraint applying to the index subtypes, or an access type whose designated type is such a type. In either case, the index constraint shall provide a discrete range for each index of the array type, and the type of each discrete range shall be the same as that of the corresponding index.

An array constraint of the first form is *compatible* with the type if, and only if, the constraint defined by each discrete range is compatible with the corresponding index subtype and the array element constraint, if present, is compatible with the element subtype of the type. If any of the discrete ranges defines a null range, any array thus constrained is a *null array*, having no elements. An array value *satisfies* an index constraint if at each index position the array value and the index constraint have the same index range. (Note, however, that assignment and certain other operations on arrays involve an implicit subtype conversion.)

If an array constraint of the second form (including the reserved word **open** in place of an index constraint) applies to a type or subtype, then the type or subtype shall be an array type or an access type whose designated type is an array type. The array constraint imposes no further constraint on the index subtypes of the array type. An array constraint of the second form is *compatible* with the type if, and only if, the array element constraint, if present, is compatible with the element subtype of the type.

The index range for each index of an array object or array subelement of a composite object is determined as follows:

a) For a variable or signal declared by an object declaration, the subtype indication of the corresponding object declaration shall define a fully constrained subtype (and thereby, the index range for each index of the array object or subelement).

b) For a constant declared by an object declaration, if the subtype of the constant defines the index range, the index range of the constant is that defined by the subtype; otherwise, the index range of the constant is the corresponding index range of the initial value..

c) For an attribute whose value is specified by an attribute specification, if the subtype of the attribute defines the index range, the index range of the value of the attribute is that defined by the subtype; otherwise, the index range of the value of the attribute is the corresponding index range of the expression given in the specification..

d) For an object designated by an access value, the index ranges are defined by the allocator that creates the designated object (see 9.3.7).

e) For an interface object of an array type, or a subelement of an interface object for which the subelement type is an array type, each index range is obtained as follows: Let the *subtype index range* be the corresponding index range of the subtype indication of the declaration of the object.

   1) If the subtype index range is defined by a constraint, the index range of the object is the subtype index range.

   2) If the subtype index range is undefined, and the interface object or subelement is associated by more than one association element or is associated by a single association element in which the formal designator is a slice name, then the direction of the index range of the object is that of the corresponding index subtype of the base type of the interface object, and the high and low bounds of the index range of the object are respectively determined from the maximum and minimum values of the indices given in the association element or elements corresponding to the interface object or subelement.

   3) If the subtype index range is undefined, and the interface object is associated in whole (see 6.5.7.1) or is a subelement that is associated individually by a single association element other than one in which the formal designator is a slice name, then the index range of the object is obtained from the association element in the following manner:

      — For an interface object or subelement whose mode is **in**, **inout** or **linkage**, if the actual part includes a conversion function or a type conversion, then the result type of that function or the type mark of the type conversion shall define a constraint for the index range corresponding to the index range of the object, and the index range of the object is obtained from that constraint; otherwise, the index range is obtained from the object or value denoted by the actual designator.

      — For an interface object or subelement whose mode is **out**, **buffer**, **inout**, or **linkage**, if the formal part includes a conversion function or a type conversion, then the parameter subtype of that function or the type mark of the type conversion shall define a constraint for the index range corresponding to the index range of the object, and the index range is obtained from that constraint; otherwise, the index range is obtained from the object denoted by the actual designator.

      For an interface object of mode **inout** or **linkage**, the index range determined by the first rule shall be identical to the index range determined by the second rule.

   For a given array interface object, or for a given array subelement of an interface object, it is an error if application of the preceding rules yields different index ranges for any corresponding array subelements of the given interface object or given subelement.

Examples:

```
type Word is array (NATURAL range <>) of BIT;
type Memory is array (NATURAL range <>) of Word (31 downto 0);

constant A_Word: Word := "10011";
   --  The index range of A_Word is 0 to 4
```



```
entity E is
   generic (ROM: Memory);
   port (Op1, Op2: in Word; Result: out Word);
end entity E;
   --  The index ranges of the generic and the ports are defined by
   --  the actuals associated with an instance bound to E; these index
   --  ranges are accessible via the predefined array attributes
   --  (see 16.2).

signal A, B: Word (1 to 4);
signal C: Word (5 downto 0);

Instance: entity E
   generic map (ROM(1 to 2) => (others => (others => '0')))
   port map (A, Op2(3 to 4) => B(1 to 2), Op2(2) => B(3),
             Result => C(3 downto 1));
   --  In this instance, the index range of ROM is 1 to 2 (matching
   --  that of the actual), the index range of Op1 is 1 to 4 (matching
   --  the index range of A), the index range of Op2 is 2 to 4, and
   --  the index range of Result is  (3 downto 1) (again matching the
   --  index range of the actual).
```

NOTE—An index constraint with a null discrete range for an index of an array subelement of a composite array type defines a null array subelement type. The array type is not necessarily a null array type. For example, given the declarations

```
type E is array (NATURAL range <>) of INTEGER;
type T is array (1 to 10) of E (1 to 0);
```

values of type T are not null arrays. Rather, they are arrays of ten elements, each of which is a null array.

5.3.2.3 Predefined array types

The predefined array types are STRING, BOOLEAN_VECTOR, BIT_VECTOR, INTEGER_VECTOR, REAL_VECTOR, and TIME_VECTOR, defined in package STANDARD in Clause 16.

The values of the predefined type STRING are one-dimensional arrays of the predefined type CHARACTER, indexed by values of the predefined subtype POSITIVE:

```
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;
type STRING is array (POSITIVE range <>) of CHARACTER;
```

The values of the predefined types BOOLEAN_VECTOR, BIT_VECTOR, INTEGER_VECTOR, REAL_VECTOR, and TIME_VECTOR, are one-dimensional arrays of the predefined types BOOLEAN, BIT, INTEGER, REAL, and TIME, respectively, indexed by values of the predefined subtype NATURAL:

```
subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;
type BOOLEAN_VECTOR is array (NATURAL range <>) of BOOLEAN;
type BIT_VECTOR is array (NATURAL range <>) of BIT;
type INTEGER_VECTOR is array (NATURAL range <>) of INTEGER;
type REAL_VECTOR is array (NATURAL range <>) of REAL;
type TIME_VECTOR is array (NATURAL range <>) of TIME;
```

NOTE—The type REAL_VECTOR is added for consistency with VHDL-AMS, defined by IEEE Std 1076.1-2007 [B10].

Examples:

```
variable MESSAGE: STRING (1 to 17) := "THIS IS A MESSAGE";

signal LOW_BYTE: BIT_VECTOR (0 to 7);

constant MONITOR_ELEMENTS: BOOLEAN_VECTOR (LOW_BYTE'RANGE)
   := (others => FALSE);

constant ELEMENT_DELAYS: TIME_VECTOR (LOW_BYTE'RANGE)
   := (others => UNIT_DELAY);

variable BUCKETS: INTEGER_VECTOR (1 to 10);
variable AVERAGES: REAL_VECTOR (1 to 10);
```

5.3.2.4 Predefined operations on array types

Given a type declaration that declares a discrete array type T (see 9.2.3), the following operations are implicitly declared immediately following the type declaration:

```
function MINIMUM (L, R: T) return T;
function MAXIMUM (L, R: T) return T;
```

The MINIMUM operation returns the value of L if L < R, or the value of R otherwise. The MAXIMUM operation returns the value of R if L < R, or the value of L otherwise. For both operations, the comparison is performed using the predefined relational operator for the type.

In addition, given a type declaration that declares a one-dimensional array type T whose elements are of a scalar type E, the following operations are implicitly declared immediately following the type declaration:

```
function MINIMUM (L: T) return E;
function MAXIMUM (L: T) return E;
```

The values returned by these operations are defined as follows.

— The MINIMUM operation returns a value that is the least of the elements of L. That is, if L is a null array, the return value is E'HIGH. Otherwise, the return value is the result of a two-parameter MINIMUM operation. The first parameter of the two-parameter MINIMUM operation is the leftmost element of L. The second parameter of the two-parameter MINIMUM operation is the result of a single-parameter MINIMUM operation with the parameter being the rightmost (L'LENGTH – 1) elements of L.

— The MAXIMUM operation returns a value that is the greatest of the elements of L. That is, if L is a null array, the return value is E'LOW. Otherwise, the return value is the result of a two-parameter MAXIMUM operation. The first parameter of the two-parameter MAXIMUM operation is the leftmost element of L. The second parameter of the two-parameter MAXIMUM operation is the result of a single-parameter MAXIMUM operation with the parameter being the rightmost (L'LENGTH – 1) elements of L.

Given a type declaration that declares a one-dimensional array type T whose element type is a character type that contains only character literals, the following operation is implicitly declared immediately following the type declaration:

```
function TO_STRING (VALUE: T) return STRING;
```


The TO_STRING operation returns the string representation (see 5.7) of the value of its actual parameter. The result type of the operation is the type STRING defined in package STANDARD.

The following operations are implicitly declared in package STD.STANDARD immediately following the declaration of the type BIT_VECTOR:

```
alias TO_BSTRING is TO_STRING [BIT_VECTOR return STRING];
alias TO_BINARY_STRING is TO_STRING [BIT_VECTOR return STRING];
function TO_OSTRING (VALUE: BIT_VECTOR) return STRING;
alias TO_OCTAL_STRING is TO_OSTRING [BIT_VECTOR return STRING];
function TO_HSTRING (VALUE: BIT_VECTOR) return STRING;
alias TO_HEX_STRING is TO_HSTRING [BIT_VECTOR return STRING];
```

These operations return strings that are the binary, octal, and hexadecimal representations, respectively, of the parameters. For the TO_OSTRING operation, the result has an uppercase octal digit corresponding to each group of three elements in the parameter value. If the length of the parameter value is not a multiple of three, then one or two '0' elements are implicitly concatenated on the left of the parameter value to yield a value that is a multiple of three in length. Similarly, for the TO_HSTRING operation, the result has an uppercase hexadecimal digit corresponding to each group of four elements in the parameter value. If the length of the parameter value is not a multiple of four, then one, two, or three '0' elements are implicitly concatenated on the left of the parameter value to yield a value that is a multiple of four in length.

5.3.3 Record types

A record type is a composite type, objects of which consist of named elements. The value of a record object is a composite value consisting of the values of its elements.

record_type_definition ::=
    record
        element_declaration
        { element_declaration }
    end record [*record_type*_simple_name]

element_declaration ::=
    identifier_list : element_subtype_definition ;

identifier_list ::= identifier { , identifier }

element_subtype_definition ::= subtype_indication

record_constraint ::=
    (record_element_constraint { , record_element_constraint })

record_element_constraint ::= *record_element*_simple_name element_constraint

A record constraint may be used to constrain a record type or subtype (see 6.3).

Each element declaration declares an element of the record type. The identifiers of all elements of a record type shall be distinct. The use of a name that denotes a record element is not allowed within the record type definition that declares the element.

An element declaration with several identifiers is equivalent to a sequence of single element declarations. Each single element declaration declares a record element whose subtype is specified by the element subtype definition.

If a simple name appears at the end of a record type declaration, it shall repeat the identifier of the type declaration in which the record type definition is included.

A record type definition creates a record type; it consists of the element declarations in the order in which they appear in the type definition.

A record type definition in which each element subtype definition denotes either an unconstrained composite subtype or a subtype that is not a composite subtype defines a record type and a name denoting that type.

A record type definition in which at least one element subtype definition denotes a partially or fully constrained composite subtype defines both a record type and a subtype of this type:

— The record type is an implicitly declared anonymous type; this type is defined by an implicit record type definition with element declarations corresponding to those of the explicit record type definition, in the same order. Each element declaration has the same identifier list as that of the corresponding element declaration in the explicit record type definition. The element subtype definition in each element declaration denotes the base type of the subtype denoted by the element subtype definition of the corresponding element declaration in the explicit record type definition.

— The record subtype is the subtype obtained by imposition of the constraints of the subtypes denoted by the element subtype definitions of the explicit record type definition as a record constraint on the record type.

If a record type definition that defines both a record type and a subtype of that type is given for a type declaration, the simple name declared by this declaration denotes the record subtype.

If a record constraint applies to a type or subtype, then the type or subtype shall be a record type or an access type whose designated type is a record type. For each record element constraint in the record constraint, the record type shall have an element with the same simple name as the record element simple name in the record element constraint. A record constraint is *compatible* with the type if, and only if, the constraint in each record element constraint is compatible with the element subtype of the corresponding element of the type.

Example:

```
type DATE is
   record
      DAY   : INTEGER range 1 to 31;
      MONTH : MONTH_NAME;
      YEAR  : INTEGER range 0 to 4000;
   end record;

type SIGNED_FXPT_COMPLEX is
   record
      RE : SIGNED_FXPT;
      IM : SIGNED_FXPT;
   end record;

signal COMPLEX_VAL: SIGNED_FXPT_COMPLEX (RE(4 downto -16),
                                         IM(4 downto -12));
```


5.4 Access types

5.4.1 General

An object declared by an object declaration is created by the elaboration of the object declaration and is denoted by a simple name or by some other form of name. In contrast, objects that are created by the evaluation of allocators (see 9.3.7) have no simple name. Access to such an object is achieved by an *access value* returned by an allocator; the access value is said to *designate* the object.

access_type_definition ::= **access** subtype_indication

For each access type, there is a literal **null** that has a null access value designating no object at all. The null value of an access type is the default initial value of the type. Other values of an access type are obtained by evaluation of a special operation of the type, called an *allocator*. Each such access value designates an object of the subtype defined by the subtype indication of the access type definition. This subtype is called the *designated subtype* and the base type of this subtype is called the *designated type*. The designated type shall not be a file type or a protected type.

An object declared to be of an access type shall be an object of class variable. An object designated by an access value is always an object of class variable.

The only form of constraint that is allowed after the name of an access type in a subtype indication is an array constraint or a record constraint. An access value belongs to a corresponding subtype of an access type either if the access value is the null value or if the value of the designated object satisfies the constraint.

Examples:

```
type ADDRESS is access MEMORY;
type BUFFER_PTR is access TEMP_BUFFER;
```

NOTE 1—An access value delivered by an allocator can be assigned to several variables of the corresponding access type. Hence, it is possible for an object created by an allocator to be designated by more than one variable of the access type. An access value can only designate an object created by an allocator; in particular, it cannot designate an object declared by an object declaration.

NOTE 2—If the type of the object designated by the access value is an array type or has a subelement that is of an array type, this object is constrained with the array bounds supplied implicitly or explicitly for the corresponding allocator.

NOTE 3—If the designated type is a composite type, it cannot have a subelement of a file type or a protected type (see 5.3.1).

5.4.2 Incomplete type declarations

The designated type of an access type can be of any type except a file type or a protected type (see 5.4.1). In particular, the type of an element of the designated type can be another access type or even the same access type. This permits mutually dependent and recursive access types. Declarations of such types require a prior incomplete type declaration for one or more types.

incomplete_type_declaration ::= **type** identifier ;

For each incomplete type declaration there shall be a corresponding full type declaration with the same identifier. This full type declaration shall occur later and immediately within the same declarative part as the incomplete type declaration to which it corresponds.

Prior to the end of the corresponding full type declaration, the only allowed use of a name that denotes a type declared by an incomplete type declaration is as the type mark in the subtype indication of an access type definition; no constraints are allowed in this subtype indication.

Example of a recursive type:

```
type CELL;  --  An incomplete type declaration.

type LINK is access CELL;

type CELL is
   record
      VALUE : INTEGER;
      SUCC  : LINK;
      PRED  : LINK;
   end record CELL;
variable HEAD: LINK := new CELL'(0, null, null);
variable \NEXT\: LINK := HEAD.SUCC;
```

Examples of mutually dependent access types:

```
type PART;  --  Incomplete type declarations.
type WIRE;

type PART_PTR is access PART;
type WIRE_PTR is access WIRE;

type PART_LIST is array (POSITIVE range <>) of PART_PTR;
type WIRE_LIST is array (POSITIVE range <>) of WIRE_PTR;

type PART_LIST_PTR is access PART_LIST;
type WIRE_LIST_PTR is access WIRE_LIST;

type PART is
   record
      PART_NAME   : STRING (1 to MAX_STRING_LEN);
      CONNECTIONS : WIRE_LIST_PTR;
   end record;

type WIRE is
   record
      WIRE_NAME : STRING (1 to MAX_STRING_LEN);
      CONNECTS  : PART_LIST_PTR;
   end record;
```

5.4.3 Allocation and deallocation of objects

An object designated by an access value is allocated by an allocator for that type. An allocator is a primary of an expression; allocators are described in 9.3.7. For each access type, a deallocation operation is implicitly declared immediately following the full type declaration for the type. This deallocation operation makes it possible to deallocate explicitly the storage occupied by a designated object.

Given the following access type declaration:


```
type AT is access T;
```

the following operation is implicitly declared immediately following the access type declaration:

```
procedure DEALLOCATE (P: inout AT);
```

Procedure DEALLOCATE takes as its single parameter a variable of the specified access type. If the value of that variable is the null value for the specified access type, then the operation has no effect. If the value of that variable is an access value that designates an object, the storage occupied by that object is returned to the system and may then be reused for subsequent object creation through the invocation of an allocator. The access parameter P is set to the null value for the specified type.

NOTE—If an access value is copied to a second variable and is then deallocated, the second variable is not set to null and thus references invalid storage.

5.5 File types

5.5.1 General

A file type definition defines a file type. File types are used to define objects representing files in the host system environment. The value of a file object is the sequence of values contained in the host system file.

file_type_definition ::= **file of** type_mark

The type mark in a file type definition defines the subtype of the values contained in the file. The type mark may denote either a fully constrained, a partially constrained, or an unconstrained subtype. The base type of this subtype shall not be a file type, an access type, a protected type, or a formal generic type. If the base type is a composite type, it shall not contain a subelement of an access type. If the base type is an array type, it shall be a one-dimensional array type whose element subtype is fully constrained. If the base type is a record type, it shall be fully constrained.

Examples:

```
file of STRING          --  Defines a file type that can contain
                        --  an indefinite number of strings of
                        --  arbitrary length.
file of NATURAL         --  Defines a file type that can contain
                        --  only nonnegative integer values.
```

NOTE—If the base type of the subtype denoted by the type mark is a composite type, it cannot have a subelement of a file type or a protected type (see 5.3.1).

5.5.2 File operations

The language implicitly defines the operations for objects of a file type. Given the following file type declaration:

type FT **is file of** TM;

where type mark TM denotes a scalar type, a record type, or a fully constrained array subtype, the following operations are implicitly declared immediately following the file type declaration:

```
procedure FILE_OPEN (file F: FT;
                     External_Name: in STRING;
```

```
                    Open_Kind: in FILE_OPEN_KIND := READ_MODE);

procedure FILE_OPEN (Status: out FILE_OPEN_STATUS;
                     file F: FT;
                     External_Name: in STRING;
                     Open_Kind: in FILE_OPEN_KIND := READ_MODE);

procedure FILE_CLOSE (file F: FT);

procedure READ (file F: FT; VALUE: out TM);

procedure WRITE (file F: FT; VALUE: in TM);

procedure FLUSH (file F: FT);

function ENDFILE (file F: FT) return BOOLEAN;
```

The FILE_OPEN procedures open an external file specified by the External_Name parameter and associate it with the file object F. If the call to FILE_OPEN is successful (see the following), the file object is said to be *open* and the file object has an *access mode* dependent on the value supplied to the Open_Kind parameter (see 16.3).

- If the value supplied to the Open_Kind parameter is READ_MODE, the access mode of the file object is *read-only*. In addition, the file object is initialized so that a subsequent READ will return the first value in the external file. Values are read from the file object in the order that they appear in the external file.
- If the value supplied to the Open_Kind parameter is WRITE_MODE, the access mode of the file object is *write-only*. In addition, the external file is made initially empty. Values written to the file object are placed in the external file in the order in which they are written.
- If the value supplied to the Open_Kind parameter is APPEND_MODE, the access mode of the file object is *write-only*. In addition, the file object is initialized so that values written to it will be added to the end of the external file in the order in which they are written.

In the second form of FILE_OPEN, the value returned through the Status parameter indicates the results of the procedure call:

- A value of OPEN_OK indicates that the call to FILE_OPEN was successful. If the call to FILE_OPEN specifies an external file that does not exist at the beginning of the call, and if the access mode of the file object passed to the call is write-only, then the external file is created.
- A value of STATUS_ERROR indicates that the file object already has an external file associated with it.
- A value of NAME_ERROR indicates that the external file does not exist (in the case of an attempt to read from the external file) or the external file cannot be created (in the case of an attempt to write or append to an external file that does not exist). This value is also returned if the external file cannot be associated with the file object for any reason.
- A value of MODE_ERROR indicates that the external file cannot be opened with the requested Open_Kind.

The first form of FILE_OPEN causes an error to occur if the second form of FILE_OPEN, when called under identical conditions, would return a Status value other than OPEN_OK.

A call to FILE_OPEN of the first form is *successful* if and only if the call does not cause an error to occur. Similarly, a call to FILE_OPEN of the second form is successful if and only if it returns a Status value of OPEN_OK.

If a file object F is associated with an external file, procedure FILE_CLOSE terminates access to the external file associated with F and closes the external file. If F is not associated with an external file, then FILE_CLOSE has no effect. In either case, the file object is no longer open after a call to FILE_CLOSE that associates the file object with the formal parameter F.

An implicit call to FILE_CLOSE exists in a subprogram body for every file object declared in the corresponding subprogram declarative part. Each such call associates a unique file object with the formal parameter F and is called whenever the corresponding subprogram completes its execution.

Procedure READ retrieves the next value from a file; it is an error if the access mode of the file object is write-only or if the file object is not open. Procedure WRITE appends a value to a file. Procedure FLUSH requests that the implementation complete the effect of all previous calls to the WRITE procedure for a file. For the WRITE and FLUSH procedures, it is an error if the access mode of the file object is read-only or if the file is not open. Function ENDFILE returns FALSE if a subsequent READ operation on an open file object whose access mode is read-only can retrieve another value from the file; otherwise, it returns TRUE. Function ENDFILE always returns TRUE for an open file object whose access mode is write-only. It is an error if ENDFILE is called on a file object that is not open.

For a file type declaration in which the type mark denotes an unconstrained or partially constrained array type, the same operations are implicitly declared, except that the READ operation is declared as follows:

```
procedure READ (file F: FT; VALUE: out TM; LENGTH: out Natural);
```

The READ operation for such a type performs the same function as the READ operation for other types, but in addition it returns a value in parameter LENGTH that specifies the actual length of the array value read by the operation. If the object associated with formal parameter VALUE is shorter than this length, then only that portion of the array value read by the operation that can be contained in the object is returned by the READ operation, and the rest of the value is lost. If the object associated with formal parameter VALUE is longer than this length, then the entire value is returned and remaining elements of the object are unaffected by the READ operation.

An error will occur when a READ operation is performed on file F if ENDFILE(F) would return TRUE at that point.

If a READ operation for a file object is executed after a FLUSH operation for a second file object and the same external file is associated with both file objects, an implementation should fulfill the request made by the FLUSH operation before retrieving a value from the file for the READ operation.

At the beginning of the execution of any file operation, the execution of the file operation *blocks* (see 14.6) until exclusive access to the file object denoted by the formal parameter F can be granted. Exclusive access to the given file object is then granted and the execution of the file operation proceeds. Once the file operation completes, exclusive access to the given file object is rescinded.

NOTE 1—An implementation may not be able to guarantee that all values written before a FLUSH operation are flushed to the external file before a subsequent READ operation to that external file, especially when the external file resides in a distributed or remote file system.

NOTE 2—Predefined package TEXTIO is provided to support formatted human-readable I/O. It defines type TEXT (a file type representing files of variable-length text strings) and type LINE (an access type that designates such strings). READ and WRITE operations are provided in package TEXTIO that append or extract data from a single line. Additional operations are provided to read or write entire lines and to determine the status of the current line or of the file itself. Package TEXTIO is defined in Clause 16.

5.6 Protected types

5.6.1 Protected type definitions

A protected type definition defines a protected type. A protected type implements instantiatiable regions of sequential statements, each of which are guaranteed exclusive access to shared data. Shared data is a set of variable objects that may be potentially accessed as a unit by multiple processes.

protected_type_definition ::=
    protected_type_declaration
  | protected_type_body

Each protected type declaration appearing immediately within a given declarative region (see 12.1) shall have exactly one corresponding protected type body appearing immediately within the same declarative region and textually subsequent to the protected type declaration. Similarly, each protected type body appearing immediately within a given declarative region shall have exactly one corresponding protected type declaration appearing immediately within the same declarative region and textually prior to the protected type body.

5.6.2 Protected type declarations

A protected type declaration declares the external interface to a protected type.

protected_type_declaration ::=
  protected
    protected_type_declarative_part
  end protected [*protected_type*_simple_name]

protected_type_declarative_part ::=
  { protected_type_declarative_item }

protected_type_declarative_item ::=
    subprogram_declaration
  | subprogram_instantiation_declaration
  | attribute_specification
  | use_clause

If a simple name appears at the end of a protected type declaration, it shall repeat the identifier of the type declaration in which the protected type definition is included.

Each subprogram specified within a given protected type declaration defines an abstract operation, called a *method*, that operates atomically and exclusively on a single object of the protected type. In addition to the object of the protected type operated on by the subprogram, parameters may be explicitly specified in the formal parameter list of the subprogram declaration of the subprogram. Such formal parameters shall not be of an access type or a file type; moreover, they shall not have a subelement that is of an access type. Additionally, in the case of a function subprogram, the return type of the function shall not be of an access type; moreover, it shall not have a subelement that is of an access type.

NOTE 1—Composite formal parameters of methods and composite return types of function methods cannot have subelements of file types (see 5.3.1).

NOTE 2—A parameter type of a method or the return type of a function method may be a formal generic type or have a subelement of a formal generic type. However, for an instance of the enclosing declaration that defines the formal generic type, a check is required that the actual generic type is neither an access type nor contains a subelement of an access type. Depending on the implementation, this check may be done during analysis of the instantiation, or it may be deferred until the design hierarchy is elaborated.

Examples:

```
type SharedCounter is protected
   procedure increment (N: Integer := 1);
   procedure decrement (N: Integer := 1);
   impure function value return Integer;
end protected SharedCounter;

type ComplexNumber is protected
   procedure extract (variable r, i: out Real);
   procedure add (variable a, b: inout ComplexNumber);
end protected ComplexNumber;

type VariableSizeBitArray is protected
   procedure add_bit (index: Positive; value: Bit);
   impure function size return Natural;
end protected VariableSizeBitArray;
```

5.6.3 Protected type bodies

A protected type body provides the implementation for a protected type.

protected_type_body ::=
    protected body
        protected_type_body_declarative_part
    end protected body [*protected_type*_simple name]

protected_type_body_declarative_part ::=
    { protected_type_body_declarative_item }

protected_type_body_declarative_item ::=
      subprogram_declaration
    | subprogram_body
    | subprogram_instantiation_declaration
    | package_declaration
    | package_body
    | package_instantiation_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause
    | group_template_declaration
    | group_declaration

Each subprogram declaration appearing in a given protected type declaration shall have a corresponding subprogram body appearing in the corresponding protected type body.

NOTE—Subprogram bodies appearing in a protected type body not lexically conformant to any of the subprogram declarations in the corresponding protected type declaration are visible only within the protected type body. Such

subprograms may have parameters that are access and file types and (in the case of functions) return types that are or contain access types.

Examples:

```
type SharedCounter is protected body

   variable counter: Integer := 0;

   procedure increment (N: Integer := 1) is
   begin
      counter := counter + N;
   end procedure increment;

   procedure decrement (N: Integer := 1) is
   begin
      counter := counter - N;
   end procedure decrement;

   impure function value return Integer is
   begin
      return counter;
   end function value;
end protected body SharedCounter;

type ComplexNumber is protected body

   variable re, im: Real;

   procedure extract (r, i: out Real) is
   begin
      r := re;
      i := im;
   end procedure extract;

   procedure add (variable a, b: inout ComplexNumber) is
      variable a_real, b_real: Real;
      variable a_imag, b_imag: Real;
   begin
      a.extract (a_real, a_imag);
      b.extract (b_real, b_imag);
      re  := a_real + b_real;
      im := a_imag + b_imag;
   end procedure add;
end protected body ComplexNumber;

type VariableSizeBitArray is protected body
   type bit_vector_access is access Bit_Vector;

   variable bit_array: bit_vector_access := null;
   variable bit_array_length: Natural := 0;

   procedure add_bit (index: Positive; value: Bit) is
      variable tmp: bit_vector_access;
```



```
   begin
      if index > bit_array_length then
         tmp := bit_array;
         bit_array := new bit_vector (1 to index);
         if tmp /= null then
            bit_array (1 to bit_array_length) := tmp.all;
            deallocate (tmp);
         end if;
         bit_array_length := index;
      end if;
      bit_array (index) := value;
   end procedure add_bit;

   impure function size return Natural is
   begin
      return bit_array_length;
   end function size;
end protected body VariableSizeBitArray;
```

5.7 String representations

The string representation of a value of a given type is a value of type STRING, defined as follows:

- For a given value of type CHARACTER, the string representation contains one element that is the given value.
- For a given value of an enumeration type other than CHARACTER, if the value is a character literal, the string representation contains a single element that is the character literal; otherwise, the string representation is the sequence of characters in the identifier that is the given value. For an extended identifier, the string representation does not include leading or trailing backslash characters, and backslash characters in the extended identifier are not doubled in the string representation.
- For a given value of an integer type, the string representation is the sequence of characters of an abstract literal without a point and whose value is the given value. The sequence of characters of the abstract literal may be preceded by a sign character with no intervening space or format effector characters.
- For a given value of a physical type, the string representation is the sequence of characters of a physical literal whose value is the given value. The sequence of characters of the physical literal may be preceded by a sign character with no intervening space or format effector characters.
- For a given value of a floating-point type, the string representation is the sequence of characters of an abstract literal that includes a point and whose value is the given value. The sequence of characters of the abstract literal may be preceded by a sign character with no intervening space or format effector characters.
- For a given value that is of a one-dimensional array type whose element type is a character type that contains only character literals, the string representation has the same length as the given value. Each element of the string representation is the same character literal as the matching element of the given value.
- For a given value that is of a composite type other than described by the preceding paragraph, there is no string representation.
- For a value of an access type, a file type, or a protected type, there is no string representation.

In each case where a string representation is defined, the index range of the string representation is not specified by this standard.

When forming the string representation for a WRITE procedure in STD.TEXTIO (see Clause 16) or for an implicitly defined TO_STRING operation, except where otherwise specified for an overloaded TO_STRING operation:

— For a value of an integer type, the abstract literal is a decimal literal and there is no exponent.

— Letters in a basic identifier are in lowercase.

— For a value of a floating-point type, when forming the string representation for a TO_STRING operation, the abstract literal is a decimal literal in standard form, consisting of a normalized mantissa and an exponent in which the sign is present and the “e” is in lowercase. The number of digits in the standard form is implementation defined. When forming the string representation for the WRITE procedure for type REAL in which the DIGITS parameter has the value 0, the string representation is as described for a TO_STRING operation. When the DIGITS parameter is non-zero, the abstract literal is a decimal literal without the exponent, as described in 16.4.

— For a value of a physical type, when forming the string representation for a TO_STRING operation, the abstract literal is a decimal literal that is an integer literal, there is no exponent, and there is a single SPACE character between the abstract literal and the unit name. If the physical type is TIME, the unit name is the simple name of the resolution limit (see 5.2.4.2); otherwise, the unit name is the simple name of the primary unit of the physical type. When forming the string representation for the WRITE procedure for type TIME, the physical literal is as described in 16.4.

— There are no insignificant leading or trailing zeros in a decimal literal.

— There is no sign preceding the string representation of a non-negative value of an integer, physical or floating-point type.

6. Declarations

6.1 General

The language defines several kinds of named entities that are declared explicitly or implicitly by declarations. Each entity's name is defined by the declaration, either as an identifier or as an operator symbol or a character literal.

There are several forms of declaration. A declaration is one of the following:

- A type declaration
- A subtype declaration
- An object declaration
- An interface declaration
- An alias declaration
- An attribute declaration
- A component declaration
- A group template declaration
- A group declaration
- An entity declaration
- A configuration declaration
- A subprogram declaration
- A subprogram instantiation declaration
- A package declaration
- A package instantiation declaration
- A primary unit
- An architecture body
- A PSL property declaration
- A PSL sequence declaration
- An enumeration literal in an enumeration type definition
- A primary unit declaration in a physical type definition
- A secondary unit declaration in a physical type definition
- An element declaration in a record type definition
- A parameter specification in a loop statement or a for generate statement
- An implicit label declaration
- A logical name in a library clause, other than a library clause that appears within a context declarative region

For each form of declaration, the language rules define a certain region of text called the *scope* of the declaration (see 12.2). Each form of declaration associates an identifier, operator symbol, or character literal with a named entity. The identifier, operator symbol, or character literal is called the *designator* of the declaration. Only within its scope, there are places where it is possible to use the designator to refer to the associated declared entity; these places are defined by the visibility rules (see 12.3). At such places the designator is said to be a *name* of the entity; the name is said to *denote* the associated entity.

This clause describes type and subtype declarations, the various kinds of object declarations, alias declarations, attribute declarations, component declarations, and group and group template declarations. The other kinds of declarations are described in Clause 3 and Clause 4.

A declaration takes effect through the process of elaboration. Elaboration of declarations is discussed in Clause 14.

PSL verification units and declarations are described in IEEE Std 1850-2005. It is an error if a property defined by a PSL property declaration does not conform to the rules for the simple subset of PSL.

6.2 Type declarations

A type declaration declares a type. Such a type is called an *explicitly declared type*.

type_declaration ::=
    full_type_declaration
  | incomplete_type_declaration

full_type_declaration ::=
  type identifier **is** type_definition ;

type_definition ::=
    scalar_type_definition
  | composite_type_definition
  | access_type_definition
  | file_type_definition
  | protected_type_definition

The types created by the elaboration of distinct type definitions are distinct types. Moreover, they are distinct from formal generic types of entity declarations, component declarations, and uninstantiated package and subprogram declarations. The elaboration of the type definition for a scalar type or a partially constrained or fully constrained composite type creates both a base type and a subtype of the base type.

The simple name declared by a type declaration denotes the declared type, unless the type declaration declares both a base type and a subtype of the base type, in which case the simple name denotes the subtype and the base type is anonymous. A type is said to be *anonymous* if it has no simple name. For explanatory purposes, this standard sometimes refers to an anonymous type by a pseudo-name, written in italics, and uses such pseudo-names at places where the syntax normally requires an identifier.

NOTE 1—Two type definitions always define two distinct types, even if they are lexically identical. Thus, the type definitions in the following two integer type declarations define distinct types:

```
type A is range 1 to 10;
type B is range 1 to 10;
```

This applies to type declarations for other classes of types as well.

NOTE 2—The various forms of type definition are described in Clause 5. Examples of type declarations are also given in that clause.

6.3 Subtype declarations

A subtype declaration declares a subtype.

subtype_declaration ::=
  subtype identifier **is** subtype_indication ;

subtype_indication ::=
    [resolution_indication] type_mark [constraint]

resolution_indication ::=
    *resolution_function*_name | (element_resolution)

element_resolution ::= array_element_resolution | record_resolution

array_element_resolution ::= resolution_indication

record_resolution ::= record_element_resolution { , record_element_resolution }

record_element_resolution ::= *record_element*_simple_name resolution_indication

type_mark ::=
      *type*_name
    | *subtype*_name

constraint ::=
      range_constraint
    | array_constraint
    | record_constraint

element_constraint ::=
      array_constraint
    | record_constraint

A type mark denotes a type or a subtype. If a type mark is the name of a type, the type mark denotes this type and also the corresponding unconstrained subtype. The base type of a type mark is, by definition, the base type of the type or subtype denoted by the type mark.

A subtype indication defines a subtype of the base type of the type mark.

A resolution indication is said to correspond to a subtype, and associates one or more resolution functions with the subtype to which it corresponds or with subelement subtypes of the subtype to which it corresponds. A resolution indication that appears in a subtype indication corresponds to the subtype defined by the subtype indication. For that resolution indication, and any resolution indications nested within it, the association of resolution functions is specified by the following rules, applied recursively:

— If a resolution indication is in the form of a resolution function name, then the named resolution function is associated with the subtype corresponding to the resolution indication.
— If a resolution indication is in the form that contains an element resolution that is an array element resolution, then the subtype corresponding to the resolution indication shall be an array subtype. The array element resolution corresponds to the element subtype of the array subtype.
— If a resolution indication is in the form that contains an element resolution that is a record resolution, then the subtype corresponding to the resolution indication shall be a record subtype. For each record element resolution in the record resolution, the record subtype shall have an element with the same simple name as the record element simple name in the record element resolution, and the resolution indication immediately following that record element simple name in the record element resolution corresponds to the element subtype of the element with that simple name in the record subtype.

If a subtype indication includes a resolution indication that associates a resolution function name with a subtype, then any signal declared to be of that subtype will be resolved, if necessary, by the named function

(see 4.6); for an overloaded function name, the meaning of the function name is determined by context (see 4.5 and 12.5). It is an error if the function does not meet the requirements of a resolution function (see 4.6). The presence of a resolution function indication has no effect on the declarations of objects other than signals or on the declarations of files, aliases, attributes, or other subtypes.

If the subtype indication does not include a constraint, the subtype is the same as that denoted by the type mark. The condition imposed by a constraint is the condition obtained after evaluation of the expressions and ranges forming the constraint. The rules defining compatibility are given for each form of constraint in the appropriate clause. These rules are such that if a constraint is compatible with a subtype, then the condition imposed by the constraint cannot contradict any condition already imposed by the subtype on its values. An error occurs if any check of compatibility fails.

The direction of a discrete subtype indication is the same as the direction of the range constraint that appears as the constraint of the subtype indication. If no constraint is present, and the type mark denotes a subtype, the direction of the subtype indication is the same as that of the denoted subtype. If no constraint is present, and the type mark denotes a type, the direction of the subtype indication is the same as that of the range used to define the denoted type. The direction of a discrete subtype is the same as the direction of its subtype indication.

A subtype indication denoting an access type, a file type, or a protected type shall not contain a resolution function. Furthermore, the only allowable constraint on a subtype indication denoting an access type is an array constraint (and then only if the designated type is an array type) or a record constraint (and then only if the designated type is a record type).

A subtype indication denoting a subtype of a file type, a protected type, or a formal generic incomplete type of an uninstantiated package or subprogram declaration shall not contain a constraint.

NOTE—A subtype declaration does not define a new type.

6.4 Objects

6.4.1 General

An *object* is a named entity that contains (has) a value of a type. An object is one of the following:

- An object declared by an object declaration (see 6.4.2)
- A loop or generate parameter (see 10.10 and 11.8)
- A formal parameter of a subprogram (see 4.2.2)
- A formal port (see 6.5.6.3 and 11.2)
- A formal generic constant (see 6.5.6.2 and 11.2)
- A local port (see 6.8)
- A local generic constant (see 6.8)
- An implicit signal GUARD defined by the guard condition of a block statement (see 11.2)

In addition, the following are objects, but are not named entities:

- An implicit signal defined by any of the predefined attributes 'DELAYED, 'STABLE, 'QUIET, and 'TRANSACTION (see 16.2)
- An element or slice of another object (see 8.3, 8.4, and 8.5)
- An object designated by a value of an access type (see 5.4.1)

There are four classes of objects: constants, signals, variables, and files. The variable class of objects also has an additional subclass: shared variables. The class of an explicitly declared object is specified by the

reserved word that shall or may appear at the beginning of the declaration of that object. For a given object of a composite type, each subelement of that object is itself an object of the same class and subclass, if any, as the given object. The value of a composite object is the aggregation of the values of its subelements.

Objects declared by object declarations are available for use within blocks, processes, subprograms, or packages. Loop and generate parameters are implicitly declared by the corresponding statement and are available for use only within that statement. Other objects, declared by interface object declarations, create channels for the communication of values between independent parts of a description.

6.4.2 Object declarations

6.4.2.1 General

An object declaration declares an object of a specified type. Such an object is called an *explicitly declared object*.

object_declaration ::=
    constant_declaration
  | signal_declaration
  | variable_declaration
  | file_declaration

An object declaration is called a *single-object declaration* if its identifier list has a single identifier; it is called a *multiple-object declaration* if the identifier list has two or more identifiers. A multiple-object declaration is equivalent to a sequence of the corresponding number of single-object declarations. For each identifier of the list, the equivalent sequence has a single-object declaration formed by this identifier, followed by a colon and by whatever appears at the right of the colon in the multiple-object declaration; the equivalent sequence is in the same order as the identifier list.

A similar equivalence applies also for interface object declarations (see 6.5.2).

NOTE—The subelements of a composite declared object are not declared objects.

6.4.2.2 Constant declarations

A constant declaration declares a *constant* of the specified type. Such a constant is an *explicitly declared constant*.

constant_declaration ::=
  constant identifier_list : subtype_indication [:= expression] ;

If the assignment symbol ":=" followed by an expression is present in a constant declaration, the expression specifies the value of the constant; the type of the expression shall be that of the constant. The value of a constant cannot be modified after the declaration is elaborated.

If the assignment symbol ":=" followed by an expression is not present in a constant declaration, then the declaration declares a deferred constant. It is an error if such a constant declaration appears anywhere other than in a package declaration. The corresponding full constant declaration, which defines the value of the constant, shall appear in the body of the package (see 4.8).

Formal parameters of subprograms that are of mode **in** may be constants, and local and formal generics may also be constants; the declarations of such objects are discussed in 6.5.2. A loop parameter is a constant within the corresponding loop (see 10.10); similarly, a generate parameter is a constant within the corresponding generate statement (see 11.8). A subelement or slice of a constant is a constant.

It is an error if a constant declaration declares a constant that is of a file type, an access type, a protected type, or a composite type that has a subelement that is of an access type.

NOTE 1—The subelements of a composite declared constant are not declared constants. Moreover, such subelements cannot be of file types or protected types (see 5.3.1).

NOTE 2—A constant may be of a formal generic type. However, for an instance of the enclosing declaration that defines the formal generic type, a check is required that the actual generic type is neither an access type nor contains a subelement of an access type. Depending on the implementation, this check may be done during analysis of the instantiation, or it may be deferred until the design hierarchy is elaborated.

Examples:

```
constant TOLER: DISTANCE := 1.5 nm;
constant PI: REAL := 3.141592;
constant CYCLE_TIME: TIME := 100 ns;
constant Propagation_Delay: DELAY_LENGTH;  -- A deferred constant.
```

6.4.2.3 Signal declarations

A signal declaration declares a *signal* of the specified type. Such a signal is an *explicitly declared signal*.

signal_declaration ::=
    signal identifier_list : subtype_indication [signal_kind] [:= expression] ;

signal_kind ::= **register** | **bus**

If a resolution indication appears in the subtype indication in the declaration of a signal or in the declaration of the subtype used to declare the signal, then each resolution function in the subtype is associated correspondingly with the declared signal or with a subelement of the declared signal. Such a signal or subelement is called a *resolved signal*.

If a signal kind appears in a signal declaration, then the signals so declared are *guarded* signals of the kind indicated. For a guarded signal that is of a composite type, each subelement is likewise a guarded signal. For a guarded signal that is of an array type, each slice (see 8.5) is likewise a guarded signal. A guarded signal may be assigned values under the control of Boolean-valued *guard conditions* (or *guards*). When a given guard becomes FALSE, the drivers of the corresponding guarded signals are implicitly assigned a null transaction (see 10.5.2.2) to cause those drivers to turn off. A disconnection specification (see 7.4) is used to specify the time required for those drivers to turn off.

If the signal declaration includes the assignment symbol followed by an expression, it shall be of the same type as the signal. Such an expression is said to be a *default expression*. The default expression defines a *default value* associated with the signal or, for a composite signal, with each scalar subelement thereof. For a signal declared to be of a scalar subtype, the value of the default expression is the default value of the signal. For a signal declared to be of a composite subtype, each scalar subelement of the value of the default expression is the default value of the corresponding subelement of the signal.

In the absence of an explicit default expression, an implicit default value is assumed for a signal of a scalar subtype or for each scalar subelement of a composite signal, each of which is itself a signal of a scalar subtype. The implicit default value for a signal of a scalar subtype T is defined to be that given by T'LEFT.

It is an error if a signal declaration declares a signal that is of a file type, an access type, a protected type, or a composite type having a subelement that is of an access type. It is also an error if a guarded signal of a scalar type is neither a resolved signal nor a subelement of a resolved signal.

A signal may have one or more *sources*. For a signal of a scalar type, each source is either a driver (see 14.7.2) or an **out**, **inout**, **buffer**, or **linkage** port of a component instance or of a block statement with which the signal is associated. For a signal of a composite type, each composite source is a collection of scalar sources, one for each scalar subelement of the signal. It is an error if, after the elaboration of a description, a signal has multiple sources and it is not a resolved signal. It is also an error if, after the elaboration of a description, a resolved signal has more sources than the number of elements in the index range of the type of the formal parameter of the resolution function associated with the resolved signal.

If a subelement or slice of a resolved signal of composite type is associated as an actual in a port map aspect (either in a component instantiation statement, a block statement, or in a binding indication), and if the corresponding formal is of mode **out**, **inout**, **buffer**, or **linkage**, then every scalar subelement of that signal shall be associated exactly once with such a formal in the same port map aspect, and the collection of the corresponding formal parts taken together constitute one source of the signal. If a resolved signal of composite type is associated as an actual in a port map aspect, that is equivalent to each of its subelements being associated in the same port map aspect.

If a subelement of a resolved signal of composite type has a driver in a given process, then every scalar subelement of that signal shall have a driver in the same process, and the collection of all of those drivers taken together constitute one source of the signal.

The default value associated with a scalar signal defines the value component of a transaction that is the initial contents of each driver (if any) of that signal. The time component of the transaction is not defined, but the transaction is understood to have already occurred by the start of simulation.

Examples:

```
signal S: STANDARD.BIT_VECTOR (1 to 10);
signal CLK1, CLK2: TIME;
signal OUTPUT: WIRED_OR MULTI_VALUED_LOGIC;
```

NOTE 1—Ports of any mode are also signals. The term *signal* is used in this standard to refer to objects declared either by signal declarations or by port declarations (or to subelements, slices, or aliases of such objects). It also refers to the implicit signal GUARD (see 11.2) and to implicit signals defined by the predefined attributes 'DELAYED, 'STABLE, 'QUIET, and 'TRANSACTION. The term *port* is used to refer to objects declared by port declarations only.

NOTE 2—Signals are given initial values by initializing their drivers. The initial values of drivers are then propagated through the corresponding net to determine the initial values of the signals that make up the net (see 14.7.3.4).

NOTE 3—The value of a signal is indirectly modified by a signal assignment statement (see 10.5); such assignments affect the future values of the signal.

NOTE 4—The subelements of a composite, declared signal are not declared signals. Moreover, such subelements cannot be of file types or protected types (see 5.3.1).

NOTE 5—A signal may be of a formal generic type. Depending on the implementation, various determinations and checks may be done during analysis of an instance of the enclosing declaration that defines the formal generic type, or they may be deferred until the design hierarchy is elaborated. These include: determining whether a signal or a subelement of a signal is resolved, based on the actual generic subtype; determining the implicit default value; checking that the actual generic type is neither an access type nor contains a subelement of an access type.

Cross-references: Disconnection specifications, 7.4; disconnection statements, 11.6; guarded assignment, 11.6; guarded blocks, 11.2; guarded targets, 11.6; signal guard, 11.2.

6.4.2.4 Variable declarations

A variable declaration declares a *variable* of the specified type. Such a variable is an *explicitly declared variable*.

variable_declaration ::=
    [**shared**] **variable** identifier_list : subtype_indication [:= expression] ;

A variable declaration that includes the reserved word **shared** is a *shared variable declaration*. A shared variable declaration declares a *shared variable*. Shared variables are a subclass of the variable class of objects. The base type of the subtype indication of a shared variable declaration shall be a protected type. Variables declared immediately within entity declarations, architecture bodies, blocks, and generate statements shall be shared variables. Variables declared immediately within subprograms and processes shall not be shared variables. Variables declared immediately within a package shall be not be shared variables if the package is declared within a subprogram, process, or protected type body; otherwise, the variables shall be shared variables. Variables declared immediately within a protected type body shall not be shared variables. Variables that appear in protected type bodies, other than within subprograms, represent shared data.

If a given variable declaration appears (directly or indirectly) within a protected type body, then the base type denoted by the subtype indication of the variable declaration shall not be the protected type defined by the protected type body.

If the variable declaration includes the assignment symbol followed by an expression, the expression specifies an initial value for the declared variable; the type of the expression shall be that of the variable. Such an expression is said to be an *initial value expression*. A variable declaration, whether it is a shared variable declaration or not, whose subtype indication denotes a protected type shall not have an initial value expression (moreover, it shall not include the immediately preceding assignment symbol).

If an initial value expression appears in the declaration of a variable, then the initial value of the variable is determined by that expression each time the variable declaration is elaborated. In the absence of an initial value expression, a default initial value applies. The default initial value for a variable of a scalar subtype T is defined to be the value given by T'LEFT. The default initial value of a variable of a composite type is defined to be the aggregate of the default initial values of all of its scalar subelements, each of which is itself a variable of a scalar subtype. The default initial value of a variable of an access type is defined to be the value **null** for that type.

NOTE 1—The value of a variable that is not a shared variable is modified by a variable assignment statement (see 10.6); such assignments take effect immediately.

NOTE 2—The variables declared within a given procedure persist until that procedure completes and returns to the caller. For procedures that contain wait statements, a variable therefore persists from one point in simulation time to another, and the value in the variable is thus maintained over time. For processes, which never complete, all variables persist from the beginning of simulation until the end of simulation.

NOTE 3—The subelements of a composite, declared variable are not declared variables.

NOTE 4—Since the language guarantees mutual exclusion of accesses to shared data, but not the order of access to such data by multiple processes in the same simulation cycle, the use of shared variables can be both non-portable and non-deterministic. For example, consider the following architecture:

```
architecture UseSharedVariables of SomeEntity is
   subtype ShortRange is INTEGER range -1 to 1;
   type ShortRangeProtected is protected
      procedure Set (V: ShortRange);
      procedure Get (V: out ShortRange);
   end protected;

   type ShortRangeProtected is protected body
      variable Local: ShortRange := 0;
      procedure Set (V: ShortRange) is
      begin
         Local := V;
      end procedure Set;
```



```
      procedure Get (V: out ShortRange) is
      begin
         V := Local;
      end procedure Get;
   end protected body;

   shared variable ShortCounter: ShortRangeProtected;

begin
   PROC1: process
      variable V: ShortRange;
   begin
      ShortCounter.Get (V);
      ShortCounter.Set (V+1);
      wait;
   end process PROC1;

   PROC2: process
      variable V: ShortRange;
   begin
      ShortCounter.Get (V);
      ShortCounter.Set (V-1);
      wait;
   end process PROC2;
end architecture UseSharedVariables;
```

In particular, the value of ShortCounter after the execution of both processes is not guaranteed to be 0.

NOTE 5—Variables that are not shared variables may have a subtype indication denoting a protected type.

NOTE 6—A variable, other than a shared variable, may be of a formal generic type. Depending on the implementation, a default initial value may be determined during analysis of an instance of the enclosing declaration that defines the formal generic type, or determination may be deferred until the design hierarchy is elaborated. A shared variable cannot be of a formal generic type, since an actual generic type shall not be a protected type.

Examples:

```
variable INDEX: INTEGER range 0 to 99 := 0;
   -- Initial value is determined by the initial value expression

variable COUNT: POSITIVE;
   -- Initial value is POSITIVE'LEFT; that is,1

variable MEMORY: BIT_MATRIX (0 to 7, 0 to 1023);
   -- Initial value is the aggregate of the initial values of each
element

shared variable Counter: SharedCounter;
   -- See 5.6.2 and 5.6.3 for the definitions of SharedCounter

shared variable addend, augend, result: ComplexNumber;
   -- See 5.6.2 and 5.6.3 for the definition of ComplexNumber

variable bit_stack: VariableSizeBitArray;
   -- See 5.6.2 and 5.6.3 for the definition of VariableSizeBitArray;
```

6.4.2.5 File declarations

A file declaration declares a *file* of the specified type. Such a file is an *explicitly declared file*.

file_declaration ::=
    file identifier_list : subtype_indication [file_open_information] ;

file_open_information ::= [**open** *file_open_kind*_expression] **is** file_logical_name

file_logical_name ::= *string*_expression

The subtype indication of a file declaration shall define a file subtype.

If file open information is included in a given file declaration, then the file declared by the declaration is opened (see 5.5.2) with an implicit call to FILE_OPEN when the file declaration is elaborated (see 14.4.2.5). This implicit call is to the FILE_OPEN procedure of the first form, and it associates the identifier with the file parameter F, the file logical name with the External_Name parameter, and the file open kind expression with the Open_Kind parameter. If a file open kind expression is not included in the file open information of a given file declaration, then the default value of READ_MODE is used during elaboration of the file declaration.

If file open information is not included in a given file declaration, then the file declared by the declaration is not opened when the file declaration is elaborated.

The file logical name shall be an expression of predefined type STRING. The value of this expression is interpreted as a logical name for a file in the host system environment. An implementation shall provide some mechanism to associate a file logical name with a host-dependent file. Such a mechanism is not defined by the language.

The file logical name identifies an external file in the host file system that is associated with the file object. This association provides a mechanism for either importing data contained in an external file into the design during simulation or exporting data generated during simulation to an external file.

If multiple file objects are associated with the same external file, and each file object has an access mode that is read-only (see 5.5.2), then values read from each file object are read from the external file associated with the file object. The language does not define the order in which such values are read from the external file, nor does it define whether each value is read once or multiple times (once per file object).

The language does not define the order of and the relationship, if any, between values read from and written to multiple file objects that are associated with the same external file. An implementation may restrict the number of file objects that are associated at one time with a given external file.

If a formal subprogram parameter is of the class **file**, it shall be associated with an actual that is a file object.

Examples:

```
type IntegerFile is file of INTEGER;

file F1: IntegerFile;
   --  No implicit FILE_OPEN is performed during elaboration.

file F2: IntegerFile is "test.dat";
   --  At elaboration, an implicit call is performed:
   --  FILE_OPEN (F2, "test.dat");
```



```
   --  The OPEN_KIND parameter defaults to READ_MODE.

file F3: IntegerFile open WRITE_MODE is "test.dat";
   --  At elaboration, an implicit call is performed:
   --  FILE_OPEN (F3, "test.dat", WRITE_MODE);
```

NOTE 1—All file objects associated with the same external file should be of the same base type.

NOTE 2—A file cannot be of a formal generic type, since an actual generic type shall not be a file type.

6.5 Interface declarations

6.5.1 General

An interface declaration is an interface object declaration, an interface type declaration, an interface subprogram declaration, or an interface package declaration.

interface_declaration ::=
    interface_object_declaration
  | interface_type_declaration
  | interface_subprogram_declaration
  | interface_package_declaration

6.5.2 Interface object declarations

An interface object declaration declares an *interface object* of a specified type. Interface objects include *interface constants* that appear as generics of a design entity, a component, a block, a package, or a subprogram, or as constant parameters of subprograms; *interface signals* that appear as ports of a design entity, component, or block, or as signal parameters of subprograms; *interface variables* that appear as variable parameters of subprograms; *interface files* that appear as file parameters of subprograms.

interface_object_declaration ::=
    interface_constant_declaration
  | interface_signal_declaration
  | interface_variable_declaration
  | interface_file_declaration

interface_constant_declaration ::=
  [**constant**] identifier_list : [**in**] subtype_indication [:= *static*_expression]

interface_signal_declaration ::=
  [**signal**] identifier_list : [mode] subtype_indication [**bus**] [:= *static*_expression]

interface_variable_declaration ::=
  [**variable**] identifier_list : [mode] subtype_indication [:= *static*_expression]

interface_file_declaration ::=
  file identifier_list : subtype_indication

mode ::= **in** | **out** | **inout** | **buffer** | **linkage**

If no mode is explicitly given in an interface declaration other than an interface file declaration, mode **in** is assumed.

For an interface constant declaration (other than a formal parameter of the predefined = or /= operator for an access type) or an interface signal declaration, the subtype indication shall define a subtype that is neither a file type, an access type, nor a protected type. Moreover, the subtype indication shall not denote a composite type with a subelement that is of an access type.

For an interface file declaration, it is an error if the subtype indication does not denote a subtype of a file type.

If an interface signal declaration includes the reserved word **bus**, then the signal declared by that interface declaration is a guarded signal of signal kind **bus**.

If an interface declaration contains a ":=" symbol followed by an expression, the expression is said to be the *default expression* of the interface object. The type of a default expression shall be that of the corresponding interface object. It is an error if a default expression appears in an interface declaration and any of the following conditions hold:

— The mode is **linkage.**

— The interface object is a formal signal parameter.

— The interface object is a formal variable parameter of mode other than **in.**

— The subtype indication of the interface declaration denotes a protected type.

In an interface signal declaration appearing in a port list, the default expression defines the default value(s) associated with the interface signal or its subelements. In the absence of a default expression, an implicit default value is assumed for the signal or for each scalar subelement, as defined for signal declarations (see 6.4.2.3). The value, whether implicitly or explicitly provided, is used to determine the initial contents of drivers, if any, of the interface signal as specified for signal declarations.

An interface object provides a channel of communication between the environment and a particular portion of a description. The value of an interface object may be determined by the value of an associated object or expression in the environment; similarly, the value of an object in the environment may be determined by the value of an associated interface object. The manner in which such associations are made is described in 6.5.7.

The value of an object is said to be *read* when one of the following conditions is satisfied:

— When the object is evaluated, and also (indirectly) when the object is associated with an interface object of the modes **in**, **inout**, or **linkage**.

— When the object is a signal and a name denoting the object appears in a sensitivity list in a wait statement or a process statement.

— When the object is a signal and the value of any of its predefined attributes 'STABLE, 'QUIET, 'DELAYED, 'TRANSACTION, 'EVENT, 'ACTIVE, 'LAST_EVENT, 'LAST_ACTIVE, or 'LAST_VALUE is read.

— When one of its subelements is read.

— When the object is a file and a READ operation is performed on the file.

— When the object is a file of type STD.TEXTIO.TEXT and the procedure STD.TEXTIO.READLINE is called with the given object associated with the formal parameter F of the given procedure.

The value of an object is said to be *updated* when one of the following conditions is satisfied:

— When it is the target of an assignment, and also (indirectly) when the object is associated with an interface object of the modes **out**, **buffer**, **inout**, or **linkage**.

— When a VHPI information model object representing the given object is updated using a call to the function `vhpi_put_value`.

— When the object is a signal and the `vhpi_schedule_transaction` function is used to schedule a transaction on a driver of the signal.

— When one of its subelements is updated.

— When the object is a file and a WRITE or FLUSH operation is performed on the file.

— When the object is a file of type STD.TEXTIO.TEXT and the procedure STD.TEXTIO.WRITELINE is called with the given object associated with the formal parameter F of the given procedure.

It is an error if an object other than a signal, variable, or file object is updated.

An interface object has one of the following modes:

— **in.** The value of the interface object is allowed to be read, but it shall not be updated by a simple waveform assignment, a conditional waveform assignment, a selected waveform assignment, a concurrent signal assignment, or a variable assignment. Reading an attribute of the interface object is allowed, unless the interface object is a subprogram signal parameter and the attribute is one of 'STABLE, 'QUIET, 'DELAYED, 'TRANSACTION, 'DRIVING, or 'DRIVING_VALUE.

— **out.** The value of the interface object is allowed to be updated and, provided it is not a signal parameter, read. Reading the attributes of the interface object is allowed, unless the interface object is a signal parameter and the attribute is one of 'STABLE, 'QUIET, 'DELAYED, 'TRANSACTION, 'EVENT, 'ACTIVE, 'LAST_EVENT, 'LAST_ACTIVE, or 'LAST_VALUE.

— **inout** or **buffer.** Reading and updating the value of the interface object is allowed. Reading the attributes of the interface object, other than the attributes 'STABLE, 'QUIET, 'DELAYED, and 'TRANSACTION of a signal parameter, is also permitted.

— **linkage.** Reading and updating the value of the interface object is allowed, but only by appearing as an actual corresponding to an interface object of mode **linkage**. No other reading or updating is permitted.

NOTE 1—A subprogram parameter that is of a file type shall be declared as a file parameter.

NOTE 2—Since shared variables are a subclass of variables, a shared variable may be associated as an actual with a formal of class variable.

NOTE 3—Ports of mode linkage are used in the Boundary Scan Description Language (see IEEE Std 1149.1™-2001 [B15]).

NOTE 4—Interface file objects do not have modes.

NOTE 5—The driving value of a port that has no source is the default value of the port (see 14.7.3.2).

NOTE 6—If the subtype indication of an interface constant declaration or an interface signal declaration denotes a composite type, the type cannot have a subelement of a file type or a protected type (see 5.3.1).

NOTE 7—Although ports of mode **out** have identical semantics to ports of mode **buffer**, there is an important design documentation distinction between them. It is intended that a port of mode **out** should be read only for passive activities, that is, for functionality used for verification purposes within monitors or property or assertion checkers. If the value of an output port is read to implement the algorithmic behavior of a description, then the port should be of mode **buffer**. Due to the potential complexity of monitors and checkers, it is not feasible to express these usage restrictions as semantic rules within the language without compromising the ability to write complex monitors and checkers.

NOTE 8—A port of mode **in** may be updated by a force assignment, a release assignment, or a call to `vhpi_put_value`. A formal parameter of mode **in** shall not be updated by a call to vhpi_put_value (see 22.5.1).

6.5.3 Interface type declarations

An interface type declaration declares an *interface type* that appears as a generic of a design entity, a component, a block, a package, or a subprogram.

interface_type_declaration ::=
    interface_incomplete_type_declaration

interface_incomplete_type_declaration ::= **type** identifier

An interface type provides a means for the environment to determine a type to be used for objects in a particular portion of a description. The set of values and applicable operations for an interface type may be determined by an associated subtype in the environment. The manner in which such associations are made is described in 6.5.7.

Within an entity declaration, an architecture body, a component declaration, or an uninstantiated subprogram or package declaration that declares a given interface type, the type declared by the given interface type declaration is distinct from the types declared by other interface type declarations and from explicitly declared types. The name of the given interface type denotes both an undefined base type and a subtype of the base type. The class (see 5.1) of the base type is not defined. The following operations are defined for the interface type:

— The basic operations of assignment, allocation, type qualification and type conversion
— The predefined equality (=) and inequality (/=) operators, implicitly declared as formal generic subprograms immediately following the interface type declaration in the enclosing interface list

The name of an interface type declaration of a block statement (including an implied block statement representing a component instance or a bound design entity), a generic-mapped package or a generic-mapped subprogram denotes the subtype specified as the corresponding actual in a generic association list.

6.5.4 Interface subprogram declarations

An interface subprogram declaration declares an *interface subprogram* that appears as a generic of a design entity, a component, a block, a package, or a subprogram.

interface_subprogram_declaration ::=
    interface_subprogram_specification [**is** interface_subprogram_default]

interface_subprogram_specification ::=
    interface_procedure_specification | interface_function_specification

interface_procedure_specification ::=
    procedure designator
        [[**parameter**] (formal_parameter_list)]

interface_function_specification ::=
    [**pure** | **impure**] **function** designator
        [[**parameter**] (formal_parameter_list)] **return** type_mark

interface_subprogram_default ::= *subprogram*_name | <>

An interface subprogram provides a means for the environment to determine a subprogram to be called in a particular portion of a description by associating an actual subprogram with the formal interface subprogram. The manner in which such associations are made is described in 6.5.7.

If an interface subprogram declaration contains an interface subprogram default in the form of a subprogram name, the subprogram name shall denote a subprogram, and the denoted subprogram and the interface subprogram shall have conforming profiles (see 4.10).

Within an entity declaration, an architecture body, a component declaration, or an uninstantiated subprogram or package declaration that declares a given interface subprogram, the name of the given interface subprogram denotes an undefined subprogram declaration and body.

The name of an interface subprogram declaration of a block statement (including an implied block statement representing a component instance or a bound design entity), a generic-mapped package or a generic-mapped subprogram denotes the subprogram specified as the corresponding actual in a generic association list.

6.5.5 Interface package declarations

An interface package declaration declares an *interface package* that appears as a generic of a design entity, a component, a block, a package, or a subprogram.

interface_package_declaration ::=
    package identifier **is new** *uninstantiated_package*_name interface_package_generic_map_aspect

interface_package_generic_map_aspect ::=
      generic_map_aspect
    | **generic map** (<>)
    | **generic map** (**default**)

An interface package provides a means for the environment to determine an instance of an uninstantiated package to be visible in a particular portion of a description by associating an actual instantiated package with the formal interface package. The manner in which such associations are made is described in 6.5.7.

The uninstantiated package name shall denote an uninstantiated package declared in a package declaration.

The interface package generic map aspect specifies the allowable actual generics of the instantiated package associated with the formal generic package (see 6.5.7.2), as follows:

- If the interface package generic map aspect is in the form of a generic map aspect, then the corresponding actual instantiated package shall have matching actual generics. Matching actual generics are described in 6.5.7.2.
- If the interface package generic map aspect is in the form that includes the box (<>) symbol, then the corresponding actual instantiated package may have any actual generics.
- If the interface package generic map aspect is in the form that includes the reserved word **default**, then every generic of the uninstantiated package shall be either a generic constant with a default expression or a generic subprogram with an interface subprogram default. The interface package generic map aspect is equivalent to an implicit interface package generic map aspect containing a generic map aspect in which each generic of the uninstantiated package is associated with the corresponding default expression or subprogram name implied by the interface subprogram default. The subprogram implied by an interface subprogram default in the form of a box (<>) symbol is a subprogram directly visible at the place of the formal generic package declaration.

Within an entity declaration, an architecture body, a component declaration, or an uninstantiated subprogram or package declaration that declares a given interface package, the name of the given interface package denotes an undefined instance of the uninstantiated package.

The name of an interface package declaration of a block statement (including an implied block statement representing a component instance or a bound design entity), a generic-mapped package or a generic-mapped subprogram denotes the instantiated package specified as the corresponding actual in a generic association list.

6.5.6 Interface lists

6.5.6.1 General

An interface list contains the interface declarations required by a subprogram, a component, a design entity, a block statement, or a package.

interface_list ::=
    interface_element { ; interface_element }

interface_element ::= interface_declaration

A *generic* interface list consists entirely of interface constant declarations, interface type declarations, interface subprogram declarations, and interface package declarations. A *port* interface list consists entirely of interface signal declarations. A *parameter* interface list may contain interface constant declarations, interface signal declarations, interface variable declarations, interface file declarations, or any combination thereof.

A name that denotes an interface object declared in a port interface list or a parameter interface list shall not appear in any interface declaration within the interface list containing the denoted interface object except to declare this object. A name that denotes an interface declaration in a generic interface list may appear in an interface declaration within the interface list containing the denoted interface declaration.

NOTE—The restriction mentioned in the previous paragraph makes the following two interface lists illegal:

```
entity E is
   port (P1: STRING; P2: STRING(P1'RANGE));                    -- Illegal
   procedure X (Y1, Y2: INTEGER; Y3: INTEGER range Y1 to Y2);  -- Illegal
end E;
```

However, the following interface lists are legal:

```
entity E is
   generic (G1: INTEGER; G2: INTEGER := G1; G3, G4, G5, G6: INTEGER);
   port (P1, P2: STRING (G3 to G4));
   procedure X (Y3: INTEGER range G5 to G6);
end E;
```

6.5.6.2 Generic clauses

Generics provide a channel for information to be communicated to a block, a package, or a subprogram from its environment. The following applies to external blocks defined by design entities, to internal blocks defined by block statements, and to packages and subprograms.

generic_clause ::=
    generic (generic_list) ;

generic_list ::= *generic*_interface_list

The generics of a block, a package, or a subprogram are defined by a generic interface list. Each interface element in such a generic interface list declares a formal generic.

The value of a generic constant may be specified by the corresponding actual in a generic association list. If no such actual is specified for a given formal generic constant (either because the formal generic is unassociated or because the actual is **open**), and if a default expression is specified for that generic, the value of this expression is the value of the generic. It is an error if no actual is specified for a given formal generic constant and no default expression is present in the corresponding interface element. It is an error if some of

the subelements of a composite formal generic constant are connected and others are either unconnected or unassociated.

The subtype denoted by a generic type is specified by the corresponding actual in a generic association list. It is an error if no such actual is specified for a given formal generic type (either because the formal generic is unassociated or because the actual is **open**).

The subprogram denoted by a generic subprogram may be specified by the corresponding actual in a generic association list. If no such actual is specified for a given formal generic subprogram (either because the formal generic is unassociated or because the actual is **open**), and if an interface subprogram default is specified for that generic, the subprogram denoted by the generic is determined as follows:

— If the interface subprogram default is in the form of a subprogram name, then the subprogram denoted by the generic is the subprogram denoted by the subprogram name.

— If the interface subprogram default is in the form of a box (<>) symbol, then there shall be a subprogram directly visible at the place of the generic association list that has the same designator as the formal and that has a conforming profile to that of the formal; the subprogram denoted by the generic is the directly visible subprogram.

It is an error if no actual is specified for a given formal generic subprogram and no interface subprogram default is present in the corresponding interface element. It is an error if the actual subprogram, whether explicitly associated or associated by default, is impure and the formal generic subprogram is pure.

A call to a formal generic subprogram uses the parameter names and default expressions defined by the declaration of the formal generic subprogram. Subtype checks and conversions for the association of actual parameters with formal parameters and for the execution of a return statement from the actual subprogram use the subtypes defined by the declaration of the actual subprogram.

The instantiated package denoted by a generic package is specified by the corresponding actual in a generic association list. It is an error if no such actual is specified for a given formal generic package (either because the formal generic is unassociated or because the actual is **open**).

NOTE—Generics may be used to control structural, dataflow, or behavioral characteristics of a block, a package, or a subprogram, or may simply be used as documentation. In particular, generics may be used to specify the size of ports; the number of subcomponents within a block; the timing characteristics of a block; or even the physical characteristics of a design such as temperature, capacitance, or location.

6.5.6.3 Port clauses

Ports provide channels for dynamic communication between a block and its environment. The following applies to both external blocks defined by design entities and to internal blocks defined by block statements, including those equivalent to component instantiation statements and generate statements (see 11.8).

port_clause ::=
    port (port_list) ;

port_list ::= *port*_interface_list

The ports of a block are defined by a port interface list. Each interface element in the port interface list declares a formal port.

To communicate with other blocks, the ports of a block can be associated with signals in the environment in which the block is used. Moreover, the ports of a block may be associated with an expression in order to provide these ports with constant driving values or with values derived from signals and other ports; such ports shall be of mode **in**. A port is itself a signal (see 6.4.2.3); thus, a formal port of a block may be

associated as an actual with a formal port of an inner block. The port, signal, or expression associated with a given formal port is called the *actual* corresponding to the formal port (see 6.5.7). The actual, if a port or signal, shall be denoted by a static name (see 8.1).

If a formal port of mode **in** is associated with an expression that is not globally static (see 9.4.1) and the formal is of an unconstrained or partially constrained composite type requiring determination of index ranges from the actual according to the rules of 5.3.2.2, then the expression shall be one of the following:

— The name of an object whose subtype is globally static

— An indexed name whose prefix is one of the members of this list

— A slice name whose prefix is one of the members of this list and whose discrete range is a globally static discrete range

— An aggregate, provided all choices are locally static and all expressions in element associations are expressions described in this list

— A function call whose return type mark denotes a globally static subtype

— A qualified expression or type conversion whose type mark denotes a globally static subtype

— An expression described in this list and enclosed in parentheses

If the actual part of a given association element for a formal port of a block is the reserved word **inertial** followed by an expression, or is an expression that is not globally static, then the given association element is equivalent to association of the port with an anonymous signal implicitly declared in the declarative region that immediately encloses the block. The signal has the same subtype as the formal port and is the target of an implicit concurrent signal assignment statement of the form

anonymous <= E;

where E is the expression in the actual part of the given association element. The concurrent signal assignment statement occurs in the same statement part as the block.

After a given description is completely elaborated (see Clause 14), if a formal port is associated with an actual that is itself a port, then the following restrictions apply depending upon the mode (see 6.5.2), if any, of the formal port:

a) For a formal port of mode **in**, the associated actual shall be a port of mode **in**, **out**, **inout**, or **buffer**. This restriction applies both to an actual that is associated as a name in the actual part of an association element and to an actual that is associated as part of an expression in the actual part of an association element.

b) For a formal port of mode **out**, the associated actual shall be a port of mode **out**, **inout**, or **buffer**.

c) For a formal port of mode **inout**, the associated actual shall be a port of mode **out**, **inout**, or **buffer**.

d) For a formal port of mode **buffer**, the associated actual shall be a port of mode **out**, **inout**, or **buffer**.

e) For a formal port of mode **linkage**, the associated actual may be a port of any mode.

If a formal port is associated with an actual port, signal, or expression, then the formal port is said to be *connected*. If a formal port is instead associated with the reserved word **open**, then the formal is said to be *unconnected*. It is an error if a port of mode **in** is unconnected (see 6.5.6.3) or unassociated (see 6.5.7.3) unless its declaration includes a default expression (see 6.5.2). It is an error if a port of any mode other than **in** is unconnected or unassociated and its type is an unconstrained or partially constrained composite type. It is an error if some of the subelements of a composite formal port are connected and others are either unconnected or unassociated.

6.5.7 Association lists

6.5.7.1 General

An association list, other than one appearing in an interface package generic map aspect (see 6.5.5), establishes correspondences between formal or local generic, port, or parameter names on the one hand and local or actual names, expressions, subtypes, subprograms, or packages on the other.

```
association_list ::=
    association_element { , association_element }

association_element ::=
    [ formal_part => ] actual_part

formal_part ::=
      formal_designator
    | function_name ( formal_designator )
    | type_mark ( formal_designator )

formal_designator ::=
      generic_name
    | port_name
    | parameter_name

actual_part ::=
      actual_designator
    | function_name ( actual_designator )
    | type_mark ( actual_designator )

actual_designator ::=
      [ inertial ] expression
    | signal_name
    | variable_name
    | file_name
    | subtype_indication
    | subprogram_name
    | instantiated_package_name
    | open
```

Each association element in an association list associates one actual designator with the corresponding interface element in the interface list of a subprogram declaration, component declaration, entity declaration, block statement, or package. The corresponding interface element is determined either by position or by name.

An association element is said to be *named* if the formal designator appears explicitly; otherwise, it is said to be *positional*. For a positional association, an actual designator at a given position in an association list corresponds to the interface element at the same position in the interface list.

Named associations can be given in any order, but if both positional and named associations appear in the same association list, then all positional associations shall occur first at their normal position. Hence once a named association is used, the rest of the association list shall use only named associations.

In the following paragraphs, the term *actual* refers to an actual designator, and the term *formal* refers to a formal designator.

The formal part of a named association element may be in the form of a function call, where the single argument of the function is the formal designator itself, if and only if the formal is an interface object, the mode of the formal is **out**, **inout**, **buffer**, or **linkage**, and if the actual is not **open**. In this case, the function name shall denote a function whose single parameter is of the type of the formal and whose result is the type of the corresponding actual. Such a *conversion function* provides for type conversion in the event that data flows from the formal to the actual.

Alternatively, the formal part of a named association element may be in the form of a type conversion, where the expression to be converted is the formal designator itself, if and only if the formal is an interface object, the mode of the formal is **out**, **inout**, **buffer**, or **linkage**, and if the actual is not **open**. In this case, the base type denoted by the type mark shall be the same as the base type of the corresponding actual. Such a type conversion provides for type conversion in the event that data flows from the formal to the actual. It is an error if the type of the formal is not closely related to the type of the actual. (See 9.3.6.)

The actual part of a (named or positional) association element corresponding to a formal interface object may have the syntactic form of a function call. This form may be interpreted either as a function call whose parameter is the actual designator, or as an expression, in which case the entire expression is the actual designator. The actual part is interpreted as a function call whose parameter is the actual designator if and only if

— The corresponding function declaration has one parameter,
— The mode of the formal corresponding to the association element is **in**, **inout,** or **linkage** and the class of the formal is not **constant**,
— The function parameter is a signal name or a variable name, and
— The function name is not preceded by the reserved word **inertial**.

Otherwise, the entire expression given by the function call is interpreted as the actual designator. In the case of a function call whose parameter is the actual designator, the type of the function parameter shall be the type of the actual and the result type shall be the type of the corresponding formal. Such a function call is interpreted as application of a conversion function that provides for type conversion in the event that data flows from the actual to the formal.

Alternatively, the actual part of a (named or positional) association element corresponding to a formal interface object may have the syntactic form of a type conversion. This form may be interpreted either as a type conversion whose operand is the actual designator, or as an expression, in which case the entire expression is the actual designator. The actual part is interpreted as a type conversion whose operand is the actual designator if and only if

— The mode of the formal corresponding to the association element is **in**, **inout,** or **linkage**, and the class of the formal is not **constant**,
— The operand is a signal name or a variable name, and
— The type mark is not preceded by the reserved word **inertial**.

Otherwise, the entire expression given by the type conversion is interpreted as the actual designator. In the case of a type conversion whose operand is the actual designator, the base type denoted by the type mark shall be the same as the base type of the corresponding formal. Such a type conversion provides for type conversion in the event that data flows from the actual to the formal. It is an error if the type of the actual is not closely related to the type of the formal.

The type of the actual (after applying the conversion function or type conversion, if present in the actual part) shall be the same as the type of the corresponding formal, if the mode of the formal is **in**, **inout**, or **linkage**, and if the actual is not **open**. Similarly, if the mode of the formal is **out**, **inout**, **buffer**, or **linkage**, and if the actual is not **open**, then the type of the formal (after applying the conversion function or type conversion, if present in the formal part) shall be the same as the corresponding actual.

For the association of signals with corresponding formal ports, association of a formal of a given composite type with an actual of the same type is equivalent to the association of each scalar subelement of the formal with the matching subelement of the actual, provided that no conversion function or type conversion is present in either the actual part or the formal part of the association element. If a conversion function or type conversion is present, then the entire formal is considered to be associated with the entire actual.

Similarly, for the association of actuals with corresponding formal subprogram parameters, association of a formal parameter of a given composite type with an actual of the same type is equivalent to the association of each scalar subelement of the formal parameter with the matching subelement of the actual. Different parameter passing mechanisms may be required in each case, but in both cases the associations will have an equivalent effect. This equivalence applies provided that no actual is accessible by more than one path (see 4.2.2.2).

A formal interface object shall be either an explicitly declared interface object or member (see 5.1) of such an interface object. In the former case, such a formal is said to be *associated in whole*. In the latter cases, named association shall be used to associate the formal and actual; the subelements of such a formal are said to be *associated individually*. Furthermore, every scalar subelement of the explicitly declared interface object shall be associated exactly once with an actual (or subelement thereof) in the same association list, and all such associations shall appear in a contiguous sequence within that association list. Each association element that associates a slice or subelement (or slice thereof) of an interface object shall identify the formal with a locally static name.

If an interface element in an interface list includes a default expression for a formal generic constant, for a formal port of any mode other than **linkage**, or for a formal variable or constant parameter of mode **in**, or an interface subprogram default for a formal generic subprogram, then any corresponding association list need not include an association element for that interface element. For an interface element that is a formal generic constant, a formal signal port, or a formal variable or constant parameter, if the association element is not included in the association list, or if the actual is **open**, then the value of the default expression is used as the actual expression or signal value in an implicit association element for that interface element. For an interface element that is a formal generic subprogram, if the association element is not included in the association list, or if the actual is **open**, then the subprogram denoted by the formal generic subprogram is determined by the interface subprogram default as described in 6.5.6.2.

It is an error if an actual of **open** is associated with a formal interface object that is associated individually. An actual of **open** counts as the single association allowed for the corresponding formal interface object, but does not supply a constant, signal, or variable (as is appropriate to the object class of the formal) to the formal.

It is an error if the reserved word **inertial** appears in an association element other than in a port map aspect.

NOTE 1—It is a consequence of these rules that, if an association element is omitted from an association list in order to make use of the default expression on the corresponding interface element, all subsequent association elements in that association list shall be named associations.

NOTE 2—Although a default expression can appear in an interface element that declares a (local or formal) port, such a default expression is not interpreted as the value of an implicit association element for that port. Instead, the value of the expression is used to determine the effective value of that port during simulation if the port is left unconnected (see 14.7.3).

NOTE 3—Named association cannot be used when invoking implicitly defined operators or predefined attributes that are functions, since the formal parameters of these operators and functions are not named (see 9.2 and 16.2).

NOTE 4—Since information flows only from the actual to the formal when the mode of the formal is **in**, and since a function call is itself an expression, the actual associated with a formal of object class constant is never interpreted as a conversion function or a type conversion converting an actual designator that is an expression. Thus, the following association element is legal:

```
Param => F (open)
```

under the conditions that Param is a constant formal and F is a function returning the same base type as that of Param and having one or more parameters, all of which may be defaulted. It is an error if a conversion function or type conversion appears in the actual part when the actual designator is **open**.

6.5.7.2 Generic map aspects

A generic map aspect, other than one appearing in an interface package generic map aspect (see 6.5.5), associates values, subtypes, subprograms, or instantiated packages with the formal generics of a block, a package, or a subprogram. The following applies to external blocks defined by design entities, to internal blocks defined by block statements, and to packages and subprograms.

generic_map_aspect ::=
    generic map (*generic*_association_list)

Both named and positional association are allowed in a generic association list.

The following definitions are used in the remainder of this subclause:

— The term *actual* refers to an actual designator that appears in an association element of a generic association list.

— The term *formal* refers to a formal designator that appears in an association element of a generic association list.

The purpose of a generic map aspect is as follows:

— A generic map aspect appearing immediately within a binding indication associates actuals with the formals of the entity declaration implied by the immediately enclosing binding indication.

— A generic map aspect appearing immediately within a component instantiation statement associates actuals with the formals of the component instantiated by the statement.

— A generic map aspect appearing immediately within a block header associates actuals with the formals defined by the same block header.

— A generic map aspect appearing immediately within a package header associates actuals with the formals defined by the same package header. This applies to a generic map aspect appearing in the package header of an explicitly declared generic-mapped package or a generic-mapped package that is equivalent to a package instantiation declaration.

— A generic map aspect appearing immediately within a subprogram header associates actuals with the formals defined by the same subprogram header. This applies to a generic map aspect appearing in the subprogram header of an explicitly declared generic-mapped subprogram or a generic-mapped subprogram that is equivalent to a subprogram instantiation declaration.

In each case, for a formal generic constant, it is an error if a scalar formal is associated with more than one actual, and it is an error if a scalar subelement of any composite formal is associated with more than one scalar subelement of an actual. Similarly, for a formal generic type, a formal generic subprogram, or a formal generic package, it is an error if the formal is associated with more than one actual.

An actual associated with a formal generic constant in a generic map aspect shall be an expression or the reserved word **open**. An actual associated with a formal generic type shall be a subtype indication. An actual associated with a formal generic subprogram shall be a name that denotes a subprogram whose profile conforms to that of the formal, or the reserved word **open**. The actual, if a predefined attribute name that denotes a function, shall be one of the predefined attributes 'IMAGE, 'VALUE, 'POS, 'VAL, 'SUCC, 'PRED, 'LEFTOF, or 'RIGHTOF.

An actual associated with a formal generic package in a generic map aspect shall be a name that denotes an instance of the uninstantiated package named in the formal generic package declaration, as follows:

a) If the formal generic package declaration includes an interface package generic map aspect in the form of a generic map aspect, then the generic map aspect of the package instantiation declaration that declares the instantiated package denoted by the actual shall match the generic map aspect of the formal generic package declaration. The two generic map aspects match if, for each generic, the corresponding associated actuals, whether explicit or implicit, match as follows:
   — Two actual generic constants match if they are the same value.
   — Two actual generic types match if they denote the same subtype; that is, if the subtypes denoted by the two actual generic types have the same base type and the same constraints. Two range constraints are the same if they have the same bounds and directions. Two array constraints are the same if they define the same index ranges and the same element subtypes. Two record constraints are the same if, for each element, the element subtypes are the same.
   — Two actual generic packages match if they denote the same instantiated package.
   — Two actual generic subprograms match if they denote the same subprogram.

b) If the formal generic package declaration includes an interface package generic map aspect in the form that includes the box (<>) symbol, then the instantiated package denoted by the actual may be any instance of the uninstantiated package named in the formal generic package declaration.

c) If the formal generic package declaration includes an interface package generic map aspect in the form that includes the reserved word **default**, then the generic map aspect of the package instantiation declaration that declares the instantiated package denoted by the actual shall match the implicit generic map aspect defined in 6.5.5.

A formal that is not associated with an actual is said to be an *unassociated* formal.

NOTE 1—A generic map aspect appearing immediately within a binding indication need not associate every formal generic constant with an actual. These formals may be left unbound so that, for example, a component configuration within a configuration declaration may subsequently bind them.

NOTE 2—A local generic (from a component declaration) or formal generic (from a package, a subprogram, a block statement or from the entity declaration of the enclosing design entity) may appear as an actual in a generic map aspect.

NOTE 3—If a formal generic constant is rebound by an incremental binding indication, the actual expression associated by the formal generic in the primary binding indication is not evaluated during the elaboration of the description.

Cross-references: Generic clauses, 6.5.6.2.

Example:

Clause 16 defines an uninstantiated package in library IEEE for fixed-point binary numbers, as follows:

```
package fixed_generic_pkg is
   generic (fixed_round_style: BOOLEAN;
            fixed_overflow_style: BOOLEAN;
            fixed_guard_bits: NATURAL;
            no_warning: BOOLEAN);
   type ufixed is array (INTEGER range <>) of STD_ULOGIC;
   type sfixed is array (INTEGER range <>) of STD_ULOGIC;
   ...
end package fixed_generic_pkg;
```

The package may be instantiated in a design unit as follows:

```
package fixed_dsp_pkg is new IEEE.fixed_generic_pkg
   generic map (fixed_rounding_style => FALSE,
                fixed_overflow_style => FALSE,
                fixed_guard_bits => 0, no_warning => TRUE);
```

An uninstantiated package defining complex numbers in which the real and imaginary parts are fixed-point binary numbers with the same index ranges can be defined as follows:

```
package fixed_complex_generic_pkg is
   generic (complex_fixed_left, complex_fixed_right: INTEGER;
            package complex_fixed_formal_pkg is
               new IEEE.fixed_generic_pkg generic map (<>));
   use complex_fixed_formal_pkg.all;
   type complex is record
      re, im : sfixed(complex_fixed_left downto complex_fixed_right);
   end record;
   function "-"  (z : complex ) return complex;
   function conj (z : complex ) return complex;
   function "+"  (l: complex; r: complex) return complex;
   function "-"  (l: complex; r: complex) return complex;
   function "*"  (l: complex; r: complex) return complex;
   function "/"  (l: complex; r: complex) return complex;
end package fixed_vector_generic_pkg;
```

This package may be instantiated to use the types and operations defined in fixed_dsp_pkg as follows:

```
package dsp_complex_pkg is new fixed_complex_generic_pkg
   generic map (complex_fixed_left => 3, complex_fixed_right => -12,
                complex_fixed_formal_pkg => fixed_dsp_pkg);
```

A further uninstantiated package defining mathematical operations on fixed-point binary numbers can be defined as follows:

```
package fixed_math_generic_pkg is
   generic (package math_fixed_formal_pkg is
               new IEEE.fixed_generic_pkg generic map (<>));
   use math_fixed_formal_pkg.all;
   function sqrt (x: sfixed) return sfixed;
   function exp  (x: sfixed) return sfixed;
   ...
end package fixed_math_generic_pkg;
```

This package, together with the complex numbers package, can be used to define an uninstantiated package that provides mathematical operations on complex numbers. Since the mathematical operations and the complex number representation depend on the fixed-point number package, an instance of the fixed-point package, together with instances of the mathematical operations and complex numbers packages that refer to the fixed-point package instance, shall be provided to the complex mathematical operations package. Thus, this package has formal generic packages as follows:

```
package fixed_complex_math_generic_pkg is
   generic (complex_math_fixed_left, complex_math_fixed_right: integer;
            package complex_math_fixed_formal_pkg is
               new IEEE.fixed_generic_pkg generic map (<>);
            package fixed_math_formal_pkg is
               new fixed_math_generic_pkg
                  generic map (math_fixed_formal_pkg =>
                                  complex_math_fixed_formal_pkg);
            package fixed_complex_formal_pkg is
               new fixed_complex_generic_pkg
```



```
                  generic map (complex_fixed_left =>
                                  complex_math_fixed_left,
                               complex_fixed_right =>
                                  complex_math_fixed_right,
                               complex_fixed_formal_pkg =>
                                  complex_math_fixed_formal_pkg));
   use complex_math_fixed_formal_pkg.all,
       fixed_math_formal_pkg.all,
       fixed_complex_formal_pkg.all;
   function "abs" (z: complex  return sfixed;
   function arg   (z: complex) return sfixed;
   function sqrt  (z: complex) return complex;
   ...
end package fixed_complex_math_generic_pkg;
```

The mathematical packages my be instantiated as follows:

```
package dsp_math_pkg is new fixed_math_generic_pkg
   generic map ( math_fixed_formal_pkg => fixed_dsp_pkg );
package dsp_complex_math_pkg is new fixed_complex_math_generic_pkg
   generic map (complex_math_fixed_left => 3,
                complex_math_fixed_right => 3,
                complex_math_fixed_formal_pkg => fixed_dsp_pkg,
                fixed_math_formal_pkg => dsp_math_pkg,
                fixed_complex_formal_pkg => dsp_complex_pkg);
```

6.5.7.3 Port map aspects

A port map aspect associates signals or values with the formal ports of a block. The following applies to both external blocks defined by design entities and to internal blocks defined by block statements.

port_map_aspect ::=
   port map (*port*_association_list)

Both named and positional association are allowed in a port association list.

The following definitions are used in the remainder of this subclause:

- The term *actual* refers to an actual designator that appears in an association element of a port association list.
- The term *formal* refers to a formal designator that appears in an association element of a port association list.

The purpose of a port aspect is as follows:

- A port map aspect appearing immediately within a binding indication associates actuals with the formals of the entity declaration implied by the immediately enclosing binding indication.
- Each scalar subelement of every local port of the component instances to which an enclosing configuration specification or component configuration applies shall be associated as an actual with at least one formal or with a scalar subelement thereof. The actuals of these associations for a given local port shall be either the entire local port or any slice or subelement (or slice thereof) of the local port. The actuals in these associations shall be locally static names.
- A port map aspect appearing immediately within a component instantiation statement associates actuals with the formals of the component instantiated by the statement.

— A port map aspect appearing immediately within a block header associates actuals with the formals defined by the same block header.

In each case, it is an error if a scalar formal is associated with more than one actual, and is is an error if a scalar subelement of any composite formal is associated with more than one scalar subelement of an actual.

An actual associated with a formal port in a port map aspect shall be a signal, an expression, or the reserved word **open**.

Certain restrictions apply to the actual associated with a formal port in a port map aspect; these restrictions are described in 6.5.6.3.

A formal that is not associated with an actual is said to be an *unassociated* formal.

Example:

```
entity Buf is
   generic (Buf_Delay: TIME := 0 ns);
   port (Input_pin: in Bit; Output_pin: out Bit);
end Buf;

architecture DataFlow of Buf is
begin
   Output_pin <= Input_pin after Buf_Delay;
end DataFlow;

entity Test_Bench is
end Test_Bench;

architecture Structure of Test_Bench is
   component Buf is
      generic (Comp_Buf_Delay: TIME);
      port (Comp_I: in Bit; Comp_O: out Bit);
   end component;
   -- A binding indication; generic and port map aspects within a
   -- binding indication associate actuals (Comp_I, etc.) with formals
   -- of the entity declaration (Input_pin, etc.):
   for UUT: Buf
      use entity Work.Buf(DataFlow)
         generic map (Buf_Delay => Comp_Buf_Delay)
         port map (Input_pin => Comp_I, Output_pin=> Comp_O);

   signal S1,S2: Bit;
begin

   -- A component instantiation statement; generic and port map aspects
   -- within a component instantiation statement associate actuals
   -- (S1, etc.) with the formals of a component (Comp_I, etc.):
   UUT: Buf
      generic map (Comp_Buf_Delay => 50 ns)
      port map (Comp_I => S1, Comp_O => S2);

   -- A block statement; generic and port map aspects within the
   -- block header of a block statement associate actuals (in this
```



```
   -- case, 4) with the formals defined in the block header:
   B: block
      generic (G: INTEGER);
      generic map (G => 4);
   begin
   end block;
end Structure;
```

NOTE—A local port (from a component declaration) or formal port (from a block statement or from the entity declaration of the enclosing design entity) may appear as an actual in a port map aspect.

Cross-references: Port clauses, 6.5.6.3.

6.6 Alias declarations

6.6.1 General

An alias declaration declares an alternate name for an existing named entity.

alias_declaration ::=
    alias alias_designator [: subtype_indication] **is** name [signature] ;

alias_designator ::= identifier | character_literal | operator_symbol

An *object alias* is an alias whose alias designator denotes an object (i.e., a constant, a variable, a signal, or a file). A *nonobject alias* is an alias whose alias designator denotes some named entity other than an object. An alias can be declared for all named entities except for labels, loop parameters, and generate parameters.

The alias designator in an alias declaration denotes the named entity specified by the name and, if present, the signature in the alias declaration. An alias of a signal denotes a signal; an alias of a variable denotes a variable; an alias of a constant denotes a constant; and an alias of a file denotes a file. Similarly, an alias of a subprogram (including an operator) denotes a subprogram, an alias of an enumeration literal denotes an enumeration literal, and so forth.

If the alias designator is a character literal, the name shall denote an enumeration literal. If the alias designator is an operator symbol, the name shall denote a function, and that function then overloads the operator symbol. In this latter case, the operator symbol and the function both shall meet the requirements of 4.5.2.

NOTE 1—Since, for example, the alias of a variable is a variable, every reference within this document to a designator (a name, character literal, or operator symbol) that requires the designator to denote a named entity with certain characteristics (e.g., to be a variable) allows the designator to denote an alias, so long as the aliased name denotes a named entity having the required characteristics. This situation holds except where aliases are specifically prohibited.

NOTE 2—The alias of an overloadable named entity is itself overloadable.

6.6.2 Object aliases

The following rules apply to object aliases:

a) A signature shall not appear in a declaration of an object alias.

b) If the name is an external name, a subtype indication shall not appear in the alias declaration.

c) The name shall be a static name (see 8.1) that denotes an object. The base type of the name specified in an alias declaration shall be the same as the base type of the type mark in the subtype indication (if the subtype indication is present). When the object denoted by the name is referenced via the alias defined by the alias declaration, the following rules apply:

1) If the subtype indication is absent
   - If the alias designator denotes a slice of an object, then the slice of the object is viewed as if it were of the subtype specified by the slice.
   - If the name is an external name, then the object is viewed as if it were of the subtype specified in the external name.
   - Otherwise, the object is viewed as if it were of the subtype specified in the declaration of the object denoted by the name.
2) If the subtype indication is present and denotes a composite subtype, then the object is viewed as if it were of the subtype specified by the subtype indication. For each index range, if any, in the subtype, if the subtype defines the index range, the object is viewed with that index range; otherwise, the object is viewed with the index range of the object. The view specified by the subtype shall include a matching element (see 9.2.3) for each element of the object denoted by the name.
3) If the subtype indication denotes a scalar subtype, then the object is viewed as if it were of the subtype specified by the subtype indication; moreover, it is an error if this subtype does not have the same bounds and direction as the subtype denoted by the object name.

d) When the prefix of an attribute name denotes the alias defined by the alias declaration, subrules 1), 2), and 3), of rule c) apply.

e) A reference to an element of an object alias is implicitly a reference to the matching element of the object denoted by the alias. A reference to a slice of an object alias consisting of the elements e_1, e_2, …, e_n is implicitly a reference to a slice of the object denoted by the alias consisting of the matching elements corresponding to each of e_1 through e_n.

6.6.3 Nonobject aliases

The following rules apply to nonobject aliases:

a) A subtype indication shall not appear in a nonobject alias.

b) A signature is required if the name denotes a subprogram (including an operator) or enumeration literal. In this case, the signature is required to match (see 4.5.3) the parameter and result type profile of exactly one of the subprograms or enumeration literals denoted by the name.

c) If the name denotes an enumeration type or a subtype of an enumeration type, then one implicit alias declaration for each of the literals of the base type immediately follows the alias declaration for the enumeration type; each such implicit declaration has, as its alias designator, the simple name or character literal of the literal and has, as its name, a name constructed by taking the name of the alias for the enumeration type or subtype and substituting the simple name or character literal being aliased for the simple name of the type or subtype. Each implicit alias has a signature that matches the parameter and result type profile of the literal being aliased.

d) Alternatively, if the name denotes a subtype of a physical type, then one implicit alias declaration for each of the units of the base type immediately follows the alias declaration for the physical type; each such implicit declaration has, as its alias designator, the simple name of the unit and has, as its name, a name constructed by taking the name of the alias for the subtype of the physical type and substituting the simple name of the unit being aliased for the simple name of the subtype.

e) Finally, if the name denotes a type or a subtype, then implicit alias declarations for each predefined operation for the type immediately follow the explicit alias declaration for the type or subtype and, if present, any implicit alias declarations for literals or units of the type. Each implicit alias has a signature that matches the parameter and result type profile of the implicit operation being aliased.

Examples:

```
variable REAL_NUMBER: BIT_VECTOR (0 to 31);
```



```
alias SIGN: BIT is REAL_NUMBER (0);
   -- SIGN is now a scalar (BIT) value

alias MANTISSA: BIT_VECTOR (23 downto 0) is REAL_NUMBER (8 to 31);
   -- MANTISSA is a 24-bit value whose range is 23 downto 0.
   -- Note that the ranges of MANTISSA and REAL_NUMBER (8 to 31)
   -- have opposite directions. A reference to MANTISSA (23 downto 18)
   -- is equivalent to a reference to REAL_NUMBER (8 to 13).

alias EXPONENT: BIT_VECTOR (1 to 7) is REAL_NUMBER (1 to 7);
   -- EXPONENT is a 7-bit value whose range is 1 to 7.

alias STD_BIT         is STD.STANDARD.BIT;  -- explicit alias

-- implicit aliases ...
-- alias '0'    is STD.STANDARD.'0' [return STD.STANDARD.BIT];
-- alias '1'    is STD.STANDARD.'1' [return STD.STANDARD.BIT];
-- alias "and"  is STD.STANDARD."and"  [STD.STANDARD.BIT,
--                                      STD.STANDARD.BIT
--                                         return STD.STANDARD.BIT];
-- alias "or"   is STD.STANDARD."or"   [STD.STANDARD.BIT,
--                                      STD.STANDARD.BIT
--                                         return STD.STANDARD.BIT];
-- alias "nand" is STD.STANDARD."nand" [STD.STANDARD.BIT,
--                                      STD.STANDARD.BIT
--                                         return STD.STANDARD.BIT];
-- alias "nor"  is STD.STANDARD."nor"  [STD.STANDARD.BIT,
--                                      STD.STANDARD.BIT
--                                         return STD.STANDARD.BIT];
-- alias "xor"  is STD.STANDARD."xor"  [STD.STANDARD.BIT,
--                                      STD.STANDARD.BIT
--                                         return STD.STANDARD.BIT];
-- alias "xnor" is STD.STANDARD."xnor" [STD.STANDARD.BIT,
--                                      STD.STANDARD.BIT
--                                         return STD.STANDARD.BIT];
-- alias "not"  is STD.STANDARD."not"  [STD.STANDARD.BIT
--                                         return STD.STANDARD.BIT];
-- alias "="    is STD.STANDARD."="    [STD.STANDARD.BIT,
--                                      STD.STANDARD.BIT
--                                         return STD.STANDARD.BOOLEAN];
-- alias "/="   is STD.STANDARD."/="   [STD.STANDARD.BIT,
--                                      STD.STANDARD.BIT
--                                         return STD.STANDARD.BOOLEAN];
-- alias "<"    is STD.STANDARD."<"    [STD.STANDARD.BIT,
--                                      STD.STANDARD.BIT
--                                         return STD.STANDARD.BOOLEAN];
-- alias "<="   is STD.STANDARD."<="   [STD.STANDARD.BIT,
--                                      STD.STANDARD.BIT
--                                         return STD.STANDARD.BOOLEAN];
-- alias ">"    is STD.STANDARD.">"    [STD.STANDARD.BIT,
--                                      STD.STANDARD.BIT
--                                         return STD.STANDARD.BOOLEAN];
-- alias ">="   is STD.STANDARD.">="   [STD.STANDARD.BIT,
--                                      STD.STANDARD.BIT
```

```
--                                              return STD.STANDARD.BOOLEAN];
-- alias MINIMUM is STD.STANDARD.MINIMUM [STD.STANDARD.BIT,
--                                        STD.STANDARD.BIT
--                                           return STD.STANDARD.BIT];
-- alias MAXIMUM is STD.STANDARD.MAXIMUM [STD.STANDARD.BIT,
--                                        STD.STANDARD.BIT
--                                           return STD.STANDARD.BIT];
```

NOTE—An alias of an explicitly declared object is not an explicitly declared object, nor is the alias of a subelement or slice of an explicitly declared object an explicitly declared object.

6.7 Attribute declarations

An attribute is a value, function, type, range, signal, or constant that may be associated with one or more named entities in a description. There are two categories of attributes: predefined attributes and user-defined attributes. Predefined attributes provide information about named entities in a description. Clause 16 contains the definition of all predefined attributes. Predefined attributes that are signals shall not be updated.

User-defined attributes are constants of arbitrary type. Such attributes are defined by an attribute declaration.

attribute_declaration ::=
    attribute identifier : type_mark ;

The identifier is said to be the *designator* of the attribute. An attribute may be associated with an entity declaration, an architecture, a configuration, a procedure, a function, a package, a type, a subtype, a constant, a signal, a variable, a component, a label, a literal, a unit, a group, or a file.

It is an error if the type mark denotes an access type, a file type, a protected type, or a composite type with a subelement that is of an access type. The denoted type or subtype need not be constrained.

Examples:

```
type COORDINATE is record X,Y: INTEGER; end record;
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;
attribute LOCATION: COORDINATE;
attribute PIN_NO: POSITIVE;
```

NOTE 1—A given named entity E will be decorated with the user-defined attribute A if and only if an attribute specification for the value of attribute A exists in the same declarative part as the declaration of E. In the absence of such a specification, an attribute name of the form E'A is illegal.

NOTE 2—A user-defined attribute is associated with the named entity denoted by the name specified in a declaration, not with the name itself. Hence, an attribute of an object can be referenced by using an alias for that object rather than the declared name of the object as the prefix of the attribute name, and the attribute referenced in such a way is the same attribute (and therefore has the same value) as the attribute referenced by using the declared name of the object as the prefix.

NOTE 3—A user-defined attribute of a port, signal, variable, or constant of some composite type is an attribute of the entire port, signal, variable, or constant, not of its elements. If it is necessary to associate an attribute with each element of some composite object, then the attribute itself can be declared to be of a composite type such that for each element of the object, there is a corresponding element of the attribute.

NOTE 4—If the type mark denotes a composite type, the type cannot have a subelement of a file type or a protected type (see 5.3.1).

6.8 Component declarations

A component declaration declares an interface to a virtual design entity that may be used in a component instantiation statement. A component configuration or a configuration specification can be used to associate a component instance with a design entity that resides in a library.

component_declaration ::=
    component identifier [**is**]
        [*local*_generic_clause]
        [*local*_port_clause]
    end component [*component*_simple_name] ;

Each interface object in the local generic clause declares a local generic. Each interface object in the local port clause declares a local port.

If a simple name appears at the end of a component declaration, it shall repeat the identifier of the component declaration.

6.9 Group template declarations

A group template declaration declares a *group template*, which defines the allowable classes of named entities that can appear in a group.

group_template_declaration ::=
    group identifier **is** (entity_class_entry_list) ;

entity_class_entry_list ::=
    entity_class_entry { , entity_class_entry }

entity_class_entry ::= entity_class [<>]

A group template is characterized by the number of entity class entries and the entity class at each position. Entity classes are described in 7.2.

An entity class entry that is an entity class defines the entity class that may appear at that position in the group type. An entity class entry that includes a box (<>) allows zero or more group constituents to appear in this position in the corresponding group declaration; such an entity class entry shall be the last one within the entity class entry list.

Examples:

```
group PIN2PIN is (signal, signal);  --  Groups of this type consist of
                                    --  two signals.
group RESOURCE is (label <>);       --  Groups of this type consist of
                                    --  any number of labels.

group DIFF_CYCLES is (group <>);    --  A group of groups.
```

6.10 Group declarations

A group declaration declares a group, a named collection of named entities. Named entities are described in 7.2.

group_declaration ::=

group identifier : *group_template*_name (group_constituent_list) ;

group_constituent_list ::= group_constituent { , group_constituent }

group_constituent ::= name | character_literal

It is an error if the class of any group constituent in the group constituent list is not the same as the class specified by the corresponding entity class entry in the entity class entry list of the group template.

A name that is a group constituent shall not be an attribute name (see 8.6). Moreover, if such a name contains a prefix, it is an error if the prefix is a function call.

If a group declaration appears within a package body, and a group constituent within that group declaration is the same as the simple name of the package body, then the group constituent denotes the package declaration and not the package body. The same rule holds for group declarations appearing within subprogram bodies containing group constituents with the same designator as that of the enclosing subprogram body.

If a group declaration contains a group constituent that denotes a variable of an access type, the group declaration declares a group incorporating the variable itself, and not the designated object, if any.

Examples:

```
group G1: RESOURCE (L1, L2);                -- A group of two labels.
group G2: RESOURCE (L3, L4, L5);            -- A group of three labels.
group C2Q: PIN2PIN (PROJECT.GLOBALS.CK, Q);
                                  --  Groups may associate named
                                  --  entities in different declarative
                                  --  parts (and regions).
group CONSTRAINT1: DIFF_CYCLES (G1, G3);    --  A group of groups.
```

6.11 PSL clock declarations

A PSL clock declaration may occur as an entity declarative item (see 3.2.3) or a block declarative item (3.3.2) and applies to certain PSL directives (if any) in the declarative region containing the PSL clock declaration. The PSL clock declaration, if any, that applies to a given PSL directive is the PSL clock declaration in the innermost declarative region containing both the given directive and a PSL clock directive. It is an error if more than one PSL clock declaration appears immediately with a given declarative region.

NOTE—A PSL clock declaration differs from other declarations in VHDL and PSL in that it does not declare a designator denoting some entity. It is more akin to a VHDL specification in that it associates additional information with PSL directives within a design. Hence, it is not listed as a declaration in 6.1. Since it is called a declaration in IEEE Std 1850-2005, it is included in this clause for ease of reference, rather than in Clause 7.

7. Specifications

7.1 General

This clause describes *specifications*, which may be used to associate additional information with a VHDL description. A specification associates additional information with a named entity that has been previously declared. There are three kinds of specifications: attribute specifications, configuration specifications, and disconnection specifications.

A specification always relates to named entities that already exist; thus a given specification shall either follow or (in certain cases) be contained within the declaration of the entity to which it relates. Furthermore, a specification shall always appear either immediately within the same declarative part as that in which the declaration of the named entity appears, or (in the case of specifications that relate to design units or the interface objects of design units, subprograms, or block statements) immediately within the declarative part associated with the declaration of the design unit, subprogram body, or block statement.

7.2 Attribute specification

An attribute specification associates a user-defined attribute with one or more named entities and defines the value of that attribute for those entities. The attribute specification is said to *decorate* the named entity.

attribute_specification ::=
    attribute attribute_designator **of** entity_specification **is** expression ;

entity_specification ::=
    entity_name_list : entity_class

entity_class ::=
      entity
    | **architecture**
    | **configuration**
    | **procedure**
    | **function**
    | **package**
    | **type**
    | **subtype**
    | **constant**
    | **signal**
    | **variable**
    | **component**
    | **label**
    | **literal**
    | **units**
    | **group**
    | **file**
    | **property**
    | **sequence**

entity_name_list ::=
      entity_designator { , entity_designator }
    | **others**
    | **all**

entity_designator ::= entity_tag [signature]

entity_tag ::= simple_name | character_literal | operator_symbol

The attribute designator shall denote an attribute. The entity name list identifies those named entities, both implicitly and explicitly defined, that inherit the attribute, described as follows:

— If a list of entity designators is supplied, then the attribute specification applies to the named entities that are denoted by those designators and are of the specified class. It is an error if any entity designator denotes no named entity of the specified class.

— If the reserved word **others** is supplied, then the attribute specification applies to named entities of the specified class that are declared in the immediately enclosing declarative part, provided that each such entity is not explicitly named in the entity name list of a previous attribute specification for the given attribute.

— If the reserved word **all** is supplied, then the attribute specification applies to all named entities of the specified class that are declared in the immediately enclosing declarative part.

An attribute specification with the entity name list **others** or **all** for a given entity class that appears in a declarative part shall be the last such specification for the given attribute for the given entity class in that declarative part. It is an error if a named entity in the specified entity class is declared in a given declarative part following such an attribute specification.

If a name in an entity name list denotes a subprogram or package, it denotes the subprogram declaration or package declaration. Subprogram and package bodies cannot be decorated.

An entity designator that denotes an alias of an object is required to denote the entire object, not a member of an object.

The entity tag of an entity designator containing a signature shall denote the name of one or more subprograms or enumeration literals. In this case, the signature shall match (see 4.5.3) the parameter and result type profile of exactly one subprogram or enumeration literal in the current declarative part: the enclosing attribute specification then decorates that subprogram or enumeration literal.

The expression specifies the value of this attribute for each of the named entities inheriting the attribute as a result of this attribute specification. The type of the expression in the attribute specification shall be the same as (or implicitly convertible to) the type mark in the corresponding attribute declaration. If the entity name list denotes an entity declaration, architecture body, configuration declaration, or an uninstantiated package that is declared as a design unit, then the expression is required to be locally static (see 9.4.1). Similarly, if the entity name list denotes a subprogram and the attribute designator denotes the 'FOREIGN attribute defined in package STANDARD, then the expression is required to be locally static.

An attribute specification for an attribute of an entity declaration, an architecture, a configuration, or a package shall appear immediately within the declarative part of that declaration. Similarly, an attribute specification for an attribute of an interface object of a design unit, subprogram, block statement, or package shall appear immediately within the declarative part of that design unit, subprogram, block statement, or package. An attribute specification for an attribute of a procedure, a function, a type, a subtype, an object (i.e., a constant, a file, a signal, or a variable), a component, literal, unit name, group, property, sequence, or a labeled entity shall appear within the declarative part in which that procedure, function, type, subtype, object, component, literal, unit name, group, property, sequence, or label, respectively, is explicitly or implicitly declared.

For a given named entity, the value of a user-defined attribute of that entity is the value specified in an attribute specification for that attribute of that entity.

It is an error if a given attribute is associated more than once with a given named entity. Similarly, it is an error if two different attributes with the same simple name (whether predefined or user-defined) are both associated with a given named entity.

An entity designator that is a character literal is used to associate an attribute with one or more character literals. An entity designator that is an operator symbol is used to associate an attribute with one or more overloaded operators.

If the entity tag is overloaded and the entity designator does not contain a signature, all named entities already declared in the current declarative part and matching the specification are decorated.

If an attribute specification appears, it shall follow the declaration of the named entity with which the attribute is associated, and it shall precede all references to that attribute of that named entity. Attribute specifications are allowed for all user-defined attributes, but are not allowed for predefined attributes.

An attribute specification may reference a named entity by using an alias for that entity in the entity name list, but such a reference counts as the single attribute specification that is allowed for a given attribute and therefore prohibits a subsequent specification that uses the declared name of the entity (or any other alias) as the entity designator.

An attribute specification whose entity designator contains no signature and identifies an overloaded subprogram or enumeration literal has the effect of associating that attribute with each of the designated overloaded subprograms or enumeration literals declared within that declarative part.

Examples:

```
attribute PIN_NO of CIN: signal is 10;
attribute PIN_NO of COUT: signal is 5;
attribute LOCATION of ADDER1: label is (10,15);
attribute LOCATION of others: label is (25,77);
attribute CAPACITANCE of all: signal is 15 pF;
attribute IMPLEMENTATION of G1: group is "74LS152";
attribute RISING_DELAY of C2Q: group is 7.2 ns;
```

NOTE 1—User-defined attributes represent local information only and cannot be used to pass information from one description to another. For instance, assume some signal X in an architecture body has some attribute A. Further, assume that X is associated with some local port L of component C. C in turn is associated with some design entity E(B), and L is associated with E's formal port P. Neither L nor P has attributes with the simple name A, unless such attributes are supplied via other attribute specifications; in this latter case, the values of P'A and X'A are not related in any way.

NOTE 2—The local ports and generics of a component declaration cannot be decorated, since component declarations lack a declarative part.

NOTE 3—If an attribute specification applies to an overloadable named entity, then declarations of additional named entities with the same simple name are allowed to occur in the current declarative part unless the aforementioned attribute specification has as its entity name list either of the reserved words **others** or **all**.

NOTE 4—Attribute specifications supplying either of the reserved words **others** or **all** never apply to the interface objects of design units, block statements, or subprograms.

NOTE 5—An attribute specification supplying either of the reserved words **others** or **all** may apply to none of the named entities in the current declarative part, in the event that none of the named entities in the current declarative part meet all of the requirements of the attribute specification.

NOTE 6—An enumeration literal is of class **literal**, not **function**.

7.3 Configuration specification

7.3.1 General

A configuration specification associates binding information with component labels representing instances of a given component declaration.

configuration_specification ::=
    simple_configuration_specification
  | compound_configuration_specification

simple_configuration_specification ::=
  for component_specification binding_indication ;
  [**end for** ;]

compound_configuration_specification ::=
  for component_specification binding_indication ;
    verification_unit_binding_indication ;
    { verification_unit_binding_indication ; }
  end for ;

component_specification ::=
  instantiation_list : *component*_name

instantiation_list ::=
    *instantiation*_label { , *instantiation*_label }
  | **others**
  | **all**

The instantiation list identifies those component instances with which binding information is to be associated, defined as follows:

- If a list of instantiation labels is supplied, then the configuration specification applies to the corresponding component instances. Such labels shall be (implicitly) declared within the immediately enclosing declarative part. It is an error if these component instances are not instances of the component declaration named in the component specification. It is also an error if any of the labels denote a component instantiation statement whose corresponding instantiated unit does not name a component.
- If the reserved word **others** is supplied, then the configuration specification applies to instances of the specified component declaration whose labels are (implicitly) declared in the immediately enclosing declarative part, provided that each such component instance is not explicitly named in the instantiation list of a previous configuration specification. This rule applies only to those component instantiation statements whose corresponding instantiated units name components.
- If the reserved word **all** is supplied, then the configuration specification applies to all instances of the specified component declaration whose labels are (implicitly) declared in the immediately enclosing declarative part. This rule applies only to those component instantiation statements whose corresponding instantiated units name components.

A configuration specification with the instantiation list **others** or **all** for a given component name that appears in a declarative part shall be the last such specification for the given component name in that declarative part.

The elaboration of a configuration specification results in the association of binding information with the labels identified by the instantiation list. A label that has binding information associated with it, specified by

a binding indication, is said to be *bound*. It is an error if the elaboration of a configuration specification results in the association of binding information with a component label that is already bound, unless the binding indication in the configuration specification is an incremental binding indication (see 7.3.2.1). It is also an error if the elaboration of a configuration specification containing an incremental binding indication results in the association of binding information with a component label that is already incrementally bound.

NOTE—A configuration specification supplying either of the reserved words **others** or **all** may apply to none of the component instances in the current declarative part. This is the case when none of the component instances in the current declarative part meet all of the requirements of the given configuration specification.

7.3.2 Binding indication

7.3.2.1 General

A binding indication associates instances of a component with a particular design entity. It may also associate actuals with formals declared in the entity declaration.

binding_indication ::=
    [**use** entity_aspect]
    [generic_map_aspect]
    [port_map_aspect]

The entity aspect of a binding indication, if present, identifies the design entity with which the instances of a component are associated. If present, the generic map aspect of a binding indication identifies the expressions, subtypes, subprograms, or instantiated packages to be associated with formal generics in the entity declaration. Similarly, the port map aspect of a binding indication identifies the signals or values to be associated with formal ports in the entity declaration.

When a binding indication is used in an explicit configuration specification, it is an error if the entity aspect is absent.

A binding indication appearing in a component configuration shall have an entity aspect unless the block corresponding to the block configuration in which the given component configuration appears has one or more configuration specifications that together configure all component instances denoted in the given component configuration. The binding indications appearing in these configuration specifications are the corresponding *primary binding indications*. A binding indication need not have an entity aspect; in that case, either or both of a generic map aspect or a port map aspect shall be present in the binding indication. Such a binding indication is an *incremental binding indication*. An incremental binding indication is used to *incrementally rebind* the ports and generic constants of the denoted instance(s) under the following conditions:

— For each formal generic constant appearing in the generic map aspect of the incremental binding indication and denoting a formal generic constant that is unassociated or associated with **open** in any of the primary binding indications, the given formal generic constant is bound to the actual with which it is associated in the generic map aspect of the incremental binding indication.

— For each formal generic constant appearing in the generic map aspect of the incremental binding indication and denoting a formal generic constant that is associated with an actual other than **open** in one of the primary binding indications, the given formal generic constant is *rebound* to the actual with which it is associated in the generic map aspect of the incremental binding indication. That is, the association given in the primary binding indication has no effect for the given instance.

— For each formal port appearing in the port map aspect of the incremental binding indication and denoting a formal port that is unassociated or associated with **open** in any of the primary binding indications, the given formal port is bound to the actual with which it is associated in the port map aspect of the incremental binding indication.

It is an error if a formal port appears in the port map aspect of the incremental binding indication and it is a formal port that is associated with an actual other than **open** in one of the primary binding indications.

If the generic map aspect or port map aspect of a primary binding indication is not present, then the default rules as described in 7.3.3 apply.

It is an error if an explicit entity aspect in an incremental binding indication does not adhere to any of the following rules:

— If the entity aspect in the corresponding primary binding indication is of the first form (fully bound), as specified in 7.3.2.2, then the entity aspect in the incremental binding indication shall also be of the first form and shall denote the same entity declaration as that of the primary binding indication. An architecture name shall be specified in the incremental binding indication if and only if the primary binding indication also identifies an architecture name; in this case, the architecture name in the incremental binding indication shall denote the same architecture name as that of the primary binding indication.

— If the entity aspect in the primary binding indication is of the second form (that is, identifying a configuration), then the entity aspect of the incremental binding indication shall be of the same form and shall denote the same configuration declaration as that of the primary binding indication.

NOTE 1—The third form (**open**) of an entity aspect does not apply to incremental binding indications as this form cannot include either a generic map aspect or a port map aspect and incremental binding indications shall contain at least one of these aspects.

NOTE 2—The entity aspect of an incremental binding indication in a component configuration is optional.

NOTE 3—The presence of an incremental binding indication will never cause the default rules of 7.3.3 to be applied.

Examples:

```
entity AND_GATE is
   generic (I1toO, I2toO: DELAY_LENGTH := 4 ns);
   port (I1, I2: in BIT; O: out BIT);
end entity AND_GATE;

entity XOR_GATE is
   generic (I1toO, I2toO: DELAY_LENGTH := 4 ns);
   port (I1, I2: in BIT; O: out BIT);
end entity XOR_GATE;

package MY_GATES is
   component AND_GATE is
      generic I1toO, I2toO: DELAY_LENGTH := 4 ns);
      port (I1, I2: in BIT; O: out BIT);
   end component AND_GATE;

   component XOR_GATE is
      generic (I1toO, I2toO: DELAY_LENGTH := 4 ns);
      port (I1, I2: in BIT; O: out BIT);
   end component XOR_GATE;
end package MY_GATES;

entity Half_Adder is
   port (X, Y: in BIT; Sum, Carry: out BIT);
end entity Half_Adder;

use WORK.MY_GATES.all;
```



```
architecture Structure of Half_Adder is
   for L1: XOR_GATE use
      entity WORK.XOR_GATE(Behavior)    --  The primary binding
         generic map (3 ns, 3 ns)       --  indication for instance L1.
         port map (I1 => I1, I2 => I2, O => O);
   for L2: AND_GATE use
      entity WORK.AND_GATE(Behavior)    --  The primary binding
         generic map (3 ns, 4 ns)       --  indication for instance L2.
         port map (I1, open, O);
begin
   L1: XOR_GATE port map (X, Y, Sum);
   L2: AND_GATE port map (X, Y, Carry);
end architecture Structure;

use WORK.GLOBAL_SIGNALS.all;
configuration Different of Half_Adder is
   for Structure
      for L1: XOR_GATE
         generic map (2.9 ns, 3.6 ns);  --  The incremental binding
      end for;                          --  indication of L1; rebinds
                                        --  its generics.
      for L2: AND_GATE
         generic map (2.8 ns, 3.25 ns)  --  The incremental binding
         port map (I2 => Tied_High);    --  indication of L2; rebinds
      end for;                          --  its generics and binds
                                        --  its open port.
   end for;
end configuration Different;
```

7.3.2.2 Entity aspect

An entity aspect identifies a particular design entity to be associated with instances of a component. An entity aspect may also specify that such a binding is to be deferred.

entity_aspect ::=
    entity *entity*_name [(*architecture*_identifier)]
  | **configuration** *configuration*_name
  | **open**

The first form of entity aspect identifies a particular entity declaration and (optionally) a corresponding architecture body. If no architecture identifier appears, then the immediately enclosing binding indication is said to *imply* the design entity whose interface is defined by the entity declaration denoted by the entity name and whose body is defined by the default binding rules for architecture identifiers (see 7.3.3). If an architecture identifier appears, then the immediately enclosing binding indication is said to *imply* the design entity consisting of the entity declaration denoted by the entity name together with an architecture body associated with the entity declaration; the architecture identifier defines a simple name that is used during the elaboration of a design hierarchy to select the appropriate architecture body. In either case, the corresponding component instances are said to be *fully bound*.

At the time of the analysis of an entity aspect of the first form, the library unit corresponding to the entity declaration denoted by the entity name is required to exist; moreover, the design unit containing the entity aspect depends on the denoted entity declaration. If the architecture identifier is also present, the library unit corresponding to the architecture identifier is required to exist only if the binding indication is part of a component configuration containing explicit block configurations or explicit component configurations;

only in this case does the design unit containing the entity aspect also depend on the denoted architecture body. In any case, the library unit corresponding to the architecture identifier is required to exist at the time that the design entity implied by the enclosing binding indication is bound to the component instance denoted by the component configuration or configuration specification containing the binding indication; if the library unit corresponding to the architecture identifier was required to exist during analysis, it is an error if the architecture identifier does not denote the same library unit as that denoted during analysis. The library unit corresponding to the architecture identifier, if it exists, shall be an architecture body associated with the entity declaration denoted by the entity name.

The second form of entity aspect identifies a design entity indirectly by identifying a configuration. In this case, the entity aspect is said to *imply* the design entity at the root of the design hierarchy that is defined by the configuration denoted by the configuration name.

At the time of the analysis of an entity aspect of the second form, the library unit corresponding to the configuration name is required to exist. The design unit containing the entity aspect depends on the configuration denoted by the configuration name.

The third form of entity aspect is used to specify that the identification of the design entity is to be deferred. In this case, the immediately enclosing binding indication is said to *not imply* any design entity. Furthermore, the immediately enclosing binding indication shall not include a generic map aspect or a port map aspect.

7.3.3 Default binding indication

In certain circumstances, a default binding indication will apply in the absence of an explicit binding indication. The default binding indication consists of a default entity aspect, together with a default generic map aspect and a default port map aspect, as appropriate.

If no visible entity declaration has the same simple name as that of the instantiated component, then the default entity aspect is **open**. A *visible entity declaration* is the first entity declaration, if any, in the following list:

a) An entity declaration that has the same simple name as that of the instantiated component and that is directly visible (see 12.3),

b) An entity declaration that has the same simple name as that of the instantiated component and that would be directly visible in the absence of a directly visible (see 12.3) component declaration with the same simple name as that of the entity declaration, or

c) An entity declaration denoted by L.C, where L is the target library and C is the simple name of the instantiated component. The *target library* is the library logical name of the library containing the design unit in which the component C is declared.

These visibility checks are made at the point of the absent explicit binding indication that causes the default binding indication to apply.

Otherwise, the default entity aspect is of the form

```
entity entity_name ( architecture_identifier )
```

where the entity name is the simple name of the instantiated component, and the architecture identifier is the same as the simple name of the most recently analyzed architecture body associated with the entity declaration. If this rule is applied either to a binding indication contained within a configuration specification or to a component configuration that does not contain an explicit inner block configuration, then the architecture identifier is determined during elaboration of the design hierarchy containing the binding indication. Likewise, if a component instantiation statement contains an instantiated unit containing

the reserved word **entity** but does not contain an explicitly specified architecture identifier, this rule is applied during the elaboration of the design hierarchy containing a component instantiation statement. In all other cases, this rule is applied during analysis of the binding indication.

It is an error if there is no architecture body associated with the entity declaration denoted by an entity name that is the simple name of the instantiated component.

The default binding indication includes a default generic map aspect if the design entity implied by the entity aspect contains formal generics. The default generic map aspect associates each local generic in the corresponding component instantiation (if any) with a formal of the same simple name. It is an error if such a formal does not exist or if its mode and type are not appropriate for such an association. Any remaining unassociated formals are associated with the actual designator **open**.

The default binding indication includes a default port map aspect if the design entity implied by the entity aspect contains formal ports. The default port map aspect associates each local port in the corresponding component instantiation (if any) with a formal of the same simple name. It is an error if such a formal does not exist or if its mode and type are not appropriate for such an association. Any remaining unassociated formals are associated with the actual designator **open**.

If an explicit binding indication lacks a generic map aspect, and if the design entity implied by the entity aspect contains formal generics, then the default generic map aspect is assumed within that binding indication. Similarly, if an explicit binding indication lacks a port map aspect, and the design entity implied by the entity aspect contains formal ports, then the default port map aspect is assumed within that binding indication.

7.3.4 Verification unit binding indication

A verification unit binding indication binds one or more PSL verification units to the design entity bound to a component instance.

verification_unit_binding_indication ::=
    use vunit verification_unit_list

verification_unit_list ::= *verification_unit*_name { , *verification_unit*_name }

Each name in a verification unit list shall denote a PSL verification unit (see 13.1 and IEEE Std 1850-2005).

It is an error if a PSL verification unit bound to a design entity by a configuration specification, whether explicit or implicit, is explicitly bound by its declaration (see IEEE Std 1850-2005). It is an error if a verification unit binding indication is specified for a component instance that is unbound or that is bound by a binding indication that has an entity aspect of the third form (**open**).

7.4 Disconnection specification

A disconnection specification defines the time delay to be used in the implicit disconnection of drivers of a guarded signal within a guarded signal assignment.

disconnection_specification ::=
    disconnect guarded_signal_specification **after** *time*_expression ;

guarded_signal_specification ::=
    *guarded*_signal_list : type_mark

signal_list ::=
    *signal*_name { , *signal*_name }
  | **others**
  | **all**

Each signal name in a signal list in a guarded signal specification shall be a locally static name that denotes a guarded signal (see 6.4.2.3). Each guarded signal shall be an explicitly declared signal or member of such a signal.

If a signal name in the guarded signal specification denotes a declared signal or a slice thereof, then the type mark in the specification shall be the same as the type mark in the subtype indication of the signal declaration (see 6.4.2.3).

If a signal name in the guarded signal specification denotes a slice of an array subelement of a composite signal, then the type mark in the specification shall be the same as the type mark in the subtype indication of the declaration of the array subelement.

If a signal name in the guarded signal specification denotes an array element of a composite signal, then the type mark in the specification shall be the same as the type mark of the element subtype indication in the declaration of the array type.

If a signal name in the guarded signal specification denotes a record element of a composite signal, then the type mark shall be the same as the type mark of the element subtype indication in the declaration of the record type.

Each signal shall either be declared in the declarative part enclosing the disconnection specification or be a member of a signal declared in that declarative part.

Subject to the aforementioned rules, a disconnection specification *applies* to the drivers of a guarded signal S specified with type mark T under the following circumstances:

— For a scalar signal S, if an explicit or implicit disconnection specification of the form

  disconnect S: T **after** *time*_expression;

  exists, then this disconnection specification applies to the drivers of S.

— For a composite signal S, an explicit or implicit disconnection specification of the form

  disconnect S: T **after** *time*_expression;

  is equivalent to a series of implicit disconnection specifications, one for each scalar subelement of the signal S. Each disconnection specification in the series is created as follows: it has, as its single signal name in its signal list, a unique scalar subelement of S. Its type mark is the same as the type of the same scalar subelement of S. Its time expression is the same as that of the original disconnection specification.

  The characteristics of the disconnection specification shall be such that each implicit disconnection specification in the series is a legal disconnection specification.

— If the signal list in an explicit or implicit disconnection specification contains more than one signal name, the disconnection specification is equivalent to a series of disconnection specifications, one for each signal name in the signal list. Each disconnection specification in the series is created as follows: It has, as its single signal name in its signal list, a unique member of the signal list from the original disconnection specification. Its type mark and time expression are the same as those in the original disconnection specification.

  The characteristics of the disconnection specification shall be such that each implicit disconnection specification in the series is a legal disconnection specification.

— An explicit disconnection specification of the form

disconnect others: T **after** *time*_expression;

is equivalent to an implicit disconnection specification where the reserved word **others** is replaced with a signal list comprised of the simple names of those guarded signals that are declared signals declared in the enclosing declarative part, whose type mark is the same as T, and that do not otherwise have an explicit disconnection specification applicable to its drivers; the remainder of the disconnection specification is otherwise unchanged. If there are no guarded signals in the enclosing declarative part whose type mark is the same as T and that do not otherwise have an explicit disconnection specification applicable to its drivers, then the preceding disconnection specification has no effect.

The characteristics of the explicit disconnection specification shall be such that the implicit disconnection specification, if any, is a legal disconnection specification.

— An explicit disconnection specification of the form

disconnect all: T **after** *time*_expression;

is equivalent to an implicit disconnection specification where the reserved word **all** is replaced with a signal list comprised of the simple names of those guarded signals that are declared signals declared in the enclosing declarative part and whose type mark is the same as T; the remainder of the disconnection specification is otherwise unchanged. If there are no guarded signals in the enclosing declarative part whose type mark is the same as T, then the preceding disconnection specification has no effect.

The characteristics of the explicit disconnection specification shall be such that the implicit disconnection specification, if any, is a legal disconnection specification.

A disconnection specification with the signal list **others** or **all** for a given type that appears in a declarative part shall be the last such specification for the given type in that declarative part. It is an error if a guarded signal of the given type is declared in a given declarative part following such a disconnection specification.

The time expression in a disconnection specification shall be static and shall evaluate to a non-negative value.

It is an error if more than one disconnection specification applies to drivers of the same signal.

If, by the aforementioned rules, no disconnection specification applies to the drivers of a guarded, scalar signal S whose type mark is T (including a scalar subelement of a composite signal), then the following default disconnection specification is implicitly assumed:

```
disconnect S : T after 0 ns;
```

A disconnection specification that applies to the drivers of a guarded signal S is the *applicable disconnection specification* for the signal S.

Thus the implicit disconnection delay for any guarded signal is always defined, either by an explicit disconnection specification or by an implicit one.

NOTE 1—A disconnection specification supplying either the reserved words **others** or **all** may apply to none of the guarded signals in the current declarative part, in the event that none of the guarded signals in the current declarative part meet all of the requirements of the disconnection specification.

NOTE 2—Since disconnection specifications are based on declarative parts, not on declarative regions, ports declared in an entity declaration cannot be referenced by a disconnection specification in a corresponding architecture body.

Cross-references: Disconnection statements, 11.6; guarded assignment, 11.6; guarded blocks, 11.2; guarded signals, 6.4.2.3; guarded targets, 11.6; signal guard, 11.2.

8. Names

8.1 General

Names can denote declared entities, whether declared explicitly or implicitly. Names can also denote the following:

- Objects denoted by access values
- Methods (see 5.6.2) of protected types
- Subelements of composite objects
- Subelements of composite values
- Slices of composite objects
- Slices of composite values
- Attributes of any named entity

```
name ::=
      simple_name
    | operator_symbol
    | character_literal
    | selected_name
    | indexed_name
    | slice_name
    | attribute_name
    | external_name

prefix ::=
      name
    | function_call
```

Certain forms of name (indexed and selected names, slice names, and attribute names) include a *prefix* that is a name or a function call. If the prefix of a name is a function call, then the name denotes an element, a slice, or an attribute, either of the result of the function call, or (if the result is an access value) of the object designated by the result. Function calls are defined in 9.3.4.

A prefix is said to be *appropriate* for a type in either of the following cases:

- The type of the prefix is the type considered.
- The type of the prefix is an access type whose designated type is the type considered.

The evaluation of a name determines the named entity denoted by the name. The evaluation of a name that has a prefix includes the evaluation of the prefix, that is, of the corresponding name or function call. If the type of the prefix is an access type, the evaluation of the prefix includes the determination of the object designated by the corresponding access value. In such a case, it is an error if the value of the prefix is a null access value. It is an error if, after all type analysis (including overload resolution), the name is ambiguous.

A name is said to be a *static name* if and only if one of the following conditions holds:

- The name is a simple name or selected name (including those that are expanded names) that does not denote a function call, an object or value of an access type, or an object of a protected type and (in the case of a selected name) whose prefix is a static name.
- The name is an indexed name whose prefix is a static name, and every expression that appears as part of the name is a static expression.

— The name is a slice name whose prefix is a static name and whose discrete range is a static discrete range.
— The name is an attribute name whose prefix is a static signal name and whose suffix is one of the predefined attributes 'DELAYED, 'STABLE, 'QUIET, or 'TRANSACTION.
— The name is an external name.

Furthermore, a name is said to be a *locally static name* if and only if one of the following conditions hold:

— The name is a simple name or selected name (including those that are expanded names) that is not an alias and that does not denote a function call, an object or value of an access type, or an object of a protected type and (in the case of a selected name) whose prefix is a locally static name.
— The name is a simple name or selected name (including those that are expanded names) that is an alias, and that the aliased name given in the corresponding alias declaration (see 6.6) is a locally static name, and (in the case of a selected name) whose prefix is a locally static name.
— The name is an indexed name whose prefix is a locally static name, and every expression that appears as part of the name is a locally static expression.
— The name is a slice name whose prefix is a locally static name and whose discrete range is a locally static discrete range.

A *static signal name* is a static name that denotes a signal. The *longest static prefix* of a signal name is the name itself, if the name is a static signal name; otherwise, it is the longest prefix of the name that is a static signal name. Similarly, a *static variable name* is a static name that denotes a variable, and the longest static prefix of a variable name is the name itself, if the name is a static variable name; otherwise, it is the longest prefix of the name that is a static variable name.

Examples:

```
S(C,2)          --A static name: C is a static constant.
R(J to 16)      --A nonstatic name: J is a signal.
                --R is the longest static prefix of R(J to 16).

T(n)            --A static name; n is a generic constant.
T(2)            --A locally static name.
```

8.2 Simple names

A simple name for a named entity is either the identifier associated with the entity by its declaration or another identifier associated with the entity by an alias declaration. In particular, the simple name for an entity declaration, a configuration, a package, a procedure, or a function is the identifier that appears in the corresponding entity declaration, configuration declaration, package declaration, procedure declaration, or function declaration, respectively. The simple name of an architecture is that defined by the identifier of the architecture body.

simple_name ::= identifier

The evaluation of a simple name has no other effect than to determine the named entity denoted by the name.

8.3 Selected names

A selected name is used to denote a named entity whose declaration appears either within the declaration of another named entity or within a design library.

selected_name ::= prefix . suffix

suffix ::=
    simple_name
  | character_literal
  | operator_symbol
  | **all**

A selected name can denote an element of a record, an object designated by an access value, or a named entity whose declaration is contained within another named entity, particularly within a library, a package, or a protected type. Furthermore, a selected name can denote all named entities whose declarations are contained within a library or a package.

For a selected name that is used to denote a record element, the suffix shall be a simple name denoting an element of a record object or value. The prefix shall be appropriate for the type of this object or value.

For a selected name that is used to denote the object designated by an access value, the suffix shall be the reserved word **all**. The prefix shall belong to an access type.

The remaining forms of selected names are called *expanded names*. The prefix of an expanded name shall not be a function call.

An expanded name denotes a primary unit contained in a design library if the prefix denotes the library and the suffix is the simple name of a primary unit whose declaration is contained in that library. An expanded name denotes all primary units contained in a library if the prefix denotes the library and the suffix is the reserved word **all**. An expanded name is not allowed for a secondary unit, particularly for an architecture body.

An expanded name denotes a named entity declared in a package if the prefix denotes the package and the suffix is the simple name, character literal, or operator symbol of a named entity whose declaration occurs immediately within that package. An expanded name denotes all named entities declared in a package if the prefix denotes the package and the suffix is the reserved word **all**.

An expanded name denotes a named entity declared immediately within a named construct if the prefix denotes a construct that is an entity declaration, an architecture body, a subprogram declaration, a subprogram body, a block statement, a process statement, a generate statement, a loop statement, or a protected type definition, and the suffix is the simple name, character literal, or operator symbol of a named entity whose declaration occurs immediately within that construct. This form of expanded name is only allowed within the construct itself, or if the prefix denotes an entity declaration and the expanded name occurs within an architecture body corresponding to the entity declaration.

An expanded name denotes a named entity declared immediately within an architecture body if the prefix denotes the entity declaration corresponding to the architecture body and the suffix is the simple name, character literal, or operator symbol of a named entity whose declaration occurs immediately within the architecture body. This form of expanded name is only allowed within the architecture body.

An expanded name denotes a named entity declared immediately within an elaborated protected type if the prefix denotes an object of the protected type and the suffix is a simple name of a method whose declaration appears immediately within the protected type declaration.

If, according to the visibility rules, there is at least one possible interpretation of the prefix of a selected name as the name of an enclosing entity declaration, architecture, subprogram, block statement, process statement, generate statement, loop statement, or protected type, or if there is at least one possible interpretation of the prefix of a selected name as the name of an object of a protected type, then the only

interpretations considered are those of the immediately preceding three paragraphs. In this case, the selected name is always interpreted as an expanded name. In particular, no interpretations of the prefix as a function call are considered.

Examples:

```
-- Given the following declarations:

type INSTR_TYPE is
   record
      OPCODE: OPCODE_TYPE;
   end record;
signal INSTRUCTION: INSTR_TYPE;

-- The name "INSTRUCTION.OPCODE" is the name of a record element.

-- Given the following declarations:

type INSTR_PTR is access INSTR_TYPE;
variable PTR: INSTR_PTR;

-- The name "PTR.all" is the name of the object designated by PTR.

-- Given the following library clause:

library TTL, CMOS;

-- The name "TTL.SN74LS221" is the name of a design unit contained in
-- a library and the name "CMOS.all" denotes all design units contained
-- in a library.

-- Given the following declaration and use clause:

library MKS;
use MKS.MEASUREMENTS, STD.STANDARD;

-- The name "MEASUREMENTS.VOLTAGE" denotes a named entity declared in
-- a package and the name "STANDARD.all" denotes all named entities
-- declared in a package.

-- Given the following process label and declarative part:

P: process
   variable DATA: INTEGER;
begin
   -- Within process P, the name "P.DATA" denotes a named entity
   -- declared in process P.
end process;

counter.increment(5);          -- See 6.4.2.4 for the definition
counter.decrement(i);          -- of "counter."
if counter.value = 0 then ... end if;

result.add(sv1, sv2);          -- See 6.4.2.4 for the definition
```



```
                                -- of "result."
bit_stack.add_bit(1, '1');      -- See 6.4.2.4 for the definition
bit_stack.add_bit(2, '1');      -- of "bit_stack."
bit_stack.add_bit(3, '0');
```

NOTE 1—The object denoted by an access value is accessed differently depending on whether the entire object or a subelement of the object is desired. If the entire object is desired, a selected name whose prefix denotes the access value and whose suffix is the reserved word **all** is used. In this case, the access value is not automatically dereferenced, since it is necessary to distinguish an access value from the object denoted by an access value.

If a subelement of the object is desired, a selected name whose prefix denotes the access value is again used; however, the suffix in this case denotes the subelement. In this case, the access value is automatically dereferenced.

These two cases are shown in the following example:

```
type rec;

type recptr is access rec;

type rec is
   record
      value  : INTEGER;
      \next\ : recptr;
   end record;

variable list1, list2: recptr;
variable recobj: rec;

list2 := list1;           -- Access values are copied;
                          -- list1 and list2 now denote the same object.
list2 := list1.\next\;    -- list2 denotes the same object as list1.\next\.
                          -- list1.\next\ is the same as list1.all.\next\.
                          -- An implicit dereference of the access value occurs before the
                          -- "\next\" element is selected.
recobj := list2.all;      -- An explicit dereference is needed here.
```

NOTE 2—Overload resolution is used to disambiguate selected names. See rules a) and c) of 12.5.

NOTE 3—If, according to the rules of this subclause and of 12.5, there is not exactly one interpretation of a selected name that satisfies these rules, then the selected name is ambiguous.

8.4 Indexed names

An indexed name denotes an element of an array.

indexed_name ::= prefix (expression { , expression })

The prefix of an indexed name shall be appropriate for an array type. The expressions specify the index values for the element; there shall be one such expression for each index position of the array, and each expression shall be of the type of the corresponding index. For the evaluation of an indexed name, the prefix and the expressions are evaluated. It is an error if an index value does not belong to the range of the corresponding index range of the array.

Examples:

```
REGISTER_ARRAY(5)          -- An element of a one-dimensional array
MEMORY_CELL(1024,7)        -- An element of a two-dimensional array
```

NOTE—If a name (including one used as a prefix) has an interpretation both as an indexed name and as a function call, then the innermost complete context is used to disambiguate the name. If, after applying this rule, there is not exactly one interpretation of the name, then the name is ambiguous. See 12.5.

8.5 Slice names

A slice name denotes a one-dimensional array composed of a sequence of consecutive elements of another one-dimensional array. A slice of a signal is a signal; a slice of a variable is a variable; a slice of a constant is a constant; a slice of a value is a value.

slice_name ::= prefix (discrete_range)

The prefix of a slice shall be appropriate for a one-dimensional array object. The base type of this array type is the type of the slice.

The bounds of the discrete range define those of the slice and shall be of the type of the index of the array. The slice is a *null slice* if the discrete range is a null range. It is an error if the direction of the discrete range is not the same as that of the index range of the array denoted by the prefix of the slice name.

For the evaluation of a name that is a slice, the prefix and the discrete range are evaluated. It is an error if either of the bounds of the discrete range does not belong to the index range of the prefixing array, unless the slice is a null slice. (The bounds of a null slice need not belong to the subtype of the index.)

Examples:

```
signal   R15:  BIT_VECTOR (0 to 31);
constant DATA: BIT_VECTOR (31 downto 0);

R15(0 to 7)             -- A slice with an ascending range.
DATA(24 downto 1)       -- A slice with a descending range.
DATA(1 downto 24)       -- A null slice.
DATA(24 to 25)          -- An error.
```

NOTE—If A is a one-dimensional array of objects, the name A(N **to** N) or A(N **downto** N) is a slice that contains one element; its type is the base type of A. On the other hand, A(N) is an element of the array A and has the corresponding element type.

8.6 Attribute names

An attribute name denotes a value, function, type, range, signal, or constant associated with a named entity.

attribute_name ::=
    prefix [signature] ' attribute_designator [(expression)]

attribute_designator ::= *attribute*_simple_name

The applicable attribute designators depend on the prefix plus the signature, if any. The meaning of the prefix of an attribute shall be determinable independently of the attribute designator and independently of the fact that it is the prefix of an attribute.

It is an error if a signature follows the prefix and the prefix does not denote a subprogram or enumeration literal, or an alias thereof. In this case, the signature is required to match (see 4.5.3) the parameter and result type profile of exactly one visible subprogram or enumeration literal, as is appropriate to the prefix.

If the attribute designator denotes a predefined attribute, the expression either shall or may appear, depending upon the definition of that attribute (see Clause 16); otherwise, it shall not be present. For an attribute that denotes a function, an expression does not appear as part of the attribute name; a parenthesized expression following the attribute designator is interpreted as part of a function call (see 9.3.4).

If the prefix of an attribute name denotes an alias, then the attribute name denotes an attribute of the aliased name and not the alias itself, except when the attribute designator denotes any of the predefined attributes 'SIMPLE_NAME, 'PATH_NAME, or 'INSTANCE_NAME. If the prefix of an attribute name denotes an alias and the attribute designator denotes any of the predefined attributes SIMPLE_NAME, 'PATH_NAME, or 'INSTANCE_NAME, then the attribute name denotes the attribute of the alias and not of the aliased name.

If the attribute designator denotes a user-defined attribute, the prefix cannot denote a subelement or a slice of an object.

NOTE—An attribute name that denotes a predefined attribute that is a function may be associated as the actual for a formal generic subprogram.

Examples:

```
REG'LEFT(1)           -- The leftmost index bound of array REG

INPUT_PIN'PATH_NAME   -- The hierarchical path name of
                      -- the port INPUT_PIN

CLK'DELAYED(5 ns)     -- The signal CLK delayed by 5 ns
```

8.7 External names

An external name denotes an object declared in the design hierarchy containing the external name.

external_name ::=
    external_constant_name
  | external_signal_name
  | external_variable_name

external_constant_name ::=
  << **constant** external_pathname : subtype_indication >>

external_signal_name ::=
  << **signal** external_pathname : subtype_indication >>

external_variable_name ::=
  << **variable** external_pathname : subtype_indication >>

external_pathname ::=
    package_pathname
  | absolute_pathname
  | relative_pathname

package_pathname ::=
  @ *library*_logical_name . *package*_simple_name . { *package*_simple_name . } *object*_simple_name

absolute_pathname ::= . partial_pathname

relative_pathname ::= { ^ . } partial_pathname

partial_pathname ::= { pathname_element . } *object*_simple_name

pathname_element ::=
    *entity*_simple_name
  | *component_instantiation*_label
  | *block*_label
  | *generate_statement*_label [(*static*_expression)]
  | *package*_simple_name

The object denoted by an external name is the object whose simple name is the object simple name of the external pathname and that is declared in the elaborated declarative region identified by the external pathname, as follows:

a) First, a declarative region is initially identified:

   1) For an absolute pathname, the root declarative region encompassing the design entity that forms the root of the design hierarchy is initially identified.

   2) For a package pathname, the library logical name shall be defined by a library clause, and the library declarative region associated with the design library denoted by the library logical name is initially identified.

   3) For a relative pathname, the innermost *concurrent region* is initially identified, where a concurrent region is defined recursively to be

      — A block declarative region (including an external block and any block equivalent to a generate statement), or

      — A package declarative region (including a generic-mapped package equivalent to a package instantiation) declared immediately within a concurrent region.

      Then, for each occurrence of a circumflex accent followed by a dot, the innermost concurrent region, other than a block declarative region of a block corresponding to a component instantiation statement, containing the previously identified declarative region replaces the previously identified declarative region as the identified declarative region. It is an error when evaluating the external name if, at any stage, there is no such containing declarative region, or if the containing declarative region is the declarative region of an uninstantiated package.

b) Second, for each package simple name in a package pathname, or for each pathname element in an absolute or relative pathname, in order, the previously identified declarative region is replaced as the identified declarative region by one of the following:

   1) For a package simple name, the declarative region of the package denoted by the package simple name in the previously identified declarative region. If the package simple name denotes a package instantiation, then the declarative region is that of the equivalent generic-mapped package.

   2) For an entity simple name, the declarative region of the external block of the design entity at the root of the design hierarchy. This form of pathname element shall only occur at a place where the previously identified declarative region is the root declarative region encompassing the design entity that forms the root of the design hierarchy.

   3) For a component instantiation label, the declarative region of the design entity bound to the component instance.

   4) For a block label, the declarative region of the block.

   5) For a generate statement label, the declarative region of the equivalent block corresponding to the generate statement. If the generate statement is a for generate statement, the pathname element shall include a static expression, the type of the expression shall be the same as the type of the generate parameter, and the value of the expression shall belong to the discrete range specified for the generate parameter. The type of the expression shall be determined by applying the rules of 12.5 to the expression considered as a complete context, using the rule that the type shall be discrete. If the type of the expression is *universal_integer* and the type of the generate

parameter is an integer type, an implicit conversion of the expression to the type of the generate parameter is assumed.

It is an error when evaluating the external name if, at any stage, a declarative region corresponding to a package name in a package pathname or to a pathname element in an absolute or relative pathname does not exist. It is an error when evaluating the external name if a package simple name in an external pathname denotes an uninstantiated package.

It is an error when evaluating an external name if the identified declarative region does not contain a declaration of an object whose simple name is the object simple name of the external pathname. It is also an error when evaluating an external name if the object denoted by an external constant name is not a constant, or if the object denoted by an external signal name is not a signal, or if the object denoted by an external variable name is not a variable. Moreover, it is an error if the base type of the object denoted by an external name is not the same as the base type of the type mark in the subtype indication of the external name.

If the subtype indication denotes a composite subtype, then the object denoted by the external name is viewed as if it were of the subtype specified by the subtype indication. For each index range, if any, in the subtype, if the subtype defines the index range, the object is viewed with that index range; otherwise, the object is viewed with the index range of the object. The view specified by the subtype shall include a matching element (see 9.2.3) for each element of the object denoted by the external name.

If the subtype indication denotes a scalar subtype, then the object denoted by the external name is viewed as if it were of the subtype specified by the subtype indication; moreover, it is an error when evaluating the external name if this subtype does not have the same bounds and direction as the subtype of the object denoted by the external name.

The evaluation of an external name has no other effect than to determine the named entity denoted by the name.

NOTE 1—A generic constant may be denoted by an external constant name, and a port may be denoted by external signal name.

NOTE 2—Since the object denoted by an external name cannot be declared within a process or subprogram, if the object is a variable, it shall be a shared variable.

NOTE 3—A declarative region corresponding to a package name or a pathname element does not exist if the name or label is not declared. It may also not exist in the case of a component instance that is unbound, or in the case of an if generate statement for which no block is generated.

NOTE 4—It is not possible to use an external name to denote the local generics or local ports of a component instantiated in a component instantiation statement.

NOTE 5—If a package has the same simple name as the entity at the root of the design entity, the external pathnames for an object in the package and an object in the design hierarchy, could, in some cases, comprise the same sequence of simple names. A package pathname starts with a different delimiter (@) from an absolute pathname (.) in order to avoid such an ambiguity.

9. Expressions

9.1 General

An expression is a formula that defines the computation of a value.

```
expression ::=
      condition_operator primary
    | logical_expression

logical_expression ::=
      relation { and relation }
    | relation { or relation }
    | relation { xor relation }
    | relation [ nand relation ]
    | relation [ nor relation ]
    | relation { xnor relation }

relation ::=
    shift_expression [ relational_operator shift_expression ]

shift_expression ::=
    simple_expression [ shift_operator simple_expression ]

simple_expression ::=
    [ sign ] term { adding_operator term }

term ::=
    factor { multiplying_operator factor }

factor ::=
      primary [ ** primary ]
    | abs primary
    | not primary
    | logical_operator primary

primary ::=
      name
    | literal
    | aggregate
    | function_call
    | qualified_expression
    | type_conversion
    | allocator
    | ( expression )
```

Each primary has a value and a type. The only names allowed as primaries are attributes that yield values and names denoting objects or values. In the case of names denoting objects other than objects of file types or protected types, the value of the primary is the value of the object. In the case of names denoting either file objects or objects of protected types, the value of the primary is the entity denoted by the name.

The type of an expression depends only upon the types of its operands and on the operators applied; for an overloaded operand or operator, the determination of the operand type, or the identification of the

overloaded operator, depends on the context (see 12.5). For each predefined operator, the operand and result types are given in the following subclause.

NOTE 1—The syntax for an expression involving logical operators allows a sequence of binary **and**, **or**, **xor**, or **xnor** operators (whether predefined or user-defined), since the corresponding predefined operations are associative. For the binary operators **nand** and **nor** (whether predefined or user-defined), however, such a sequence is not allowed, since the corresponding predefined operations are not associative.

NOTE 2—The syntax for an expression involving a unary condition operator or unary logical operator in combination with any other operator requires that the unary operator and its operand be a parenthesized expression. For example, the expressions "(**and** A) **and** B" and "A **and** (**and** B)" are legal, whereas the expression "**and** A **and** B" and "A **and and** B" are not. Similarly, "**and** (**and** A)" is legal, whereas "**and and** A" is not. An expression consisting only of a unary condition oprator or unary logical operator and its operand need not be parenthesized.

NOTE 3—PSL extends the grammar of VHDL expressions to allow PSL expressions, PSL built-in function calls, and PSL union expressions as subexpressions. Such extended expressions can only appear in a VHDL description within PSL declarations and PSL directives, or in a verification unit.

9.2 Operators

9.2.1 General

The operators that may be used in expressions are defined as follows. Each operator belongs to a class of operators, all of which have the same precedence level; the classes of operators are listed in order of increasing precedence.

condition_operator ::= ??

logical_operator ::= **and** | **or** | **nand** | **nor** | **xor** | **xnor**

relational_operator ::= = | /= | < | <= | > | >= | ?= | ?/= | ?< | ?<= | ?> | ?>=

shift_operator ::= **sll** | **srl** | **sla** | **sra** | **rol** | **ror**

adding_operator ::= + | – | &

sign ::= + | –

multiplying_operator ::= * | / | **mod** | **rem**

miscellaneous_operator ::= ** | **abs** | **not**

Operators of higher precedence are associated with their operands before operators of lower precedence. Where the language allows a sequence of operators, operators with the same precedence level are associated with their operands in textual order, from left to right. The precedence of an operator is fixed and cannot be changed by the user, but parentheses can be used to control the association of operators and operands.

In general, operands in an expression are evaluated before being associated with operators. For certain operations, however, the right-hand operand is evaluated if and only if the left-hand operand has a certain value. These operations are called *short-circuit* operations. The binary logical operations **and**, **or**, **nand**, and **nor** defined for operands of types BIT and BOOLEAN are all short-circuit operations; furthermore, these are the only short-circuit operations.

Every predefined operator and every predefined MINIMUM and MAXIMUM operation is a pure function (see 4.2.1). No predefined operators have named formal parameters; therefore, named association (see 6.5.7.1) cannot be used when invoking a predefined operator.

NOTE 1—The predefined operators for the standard types are declared in package STANDARD as shown in 16.3.

NOTE 2—The operator **not** is classified as a miscellaneous operator for the purposes of defining precedence, but is otherwise classified as a logical operator.

9.2.2 Logical operators

The binary logical operators **and**, **or**, **nand**, **nor**, **xor**, and **xnor**, and the unary logical operator **not** are defined for predefined types BIT and BOOLEAN. They are also defined for any one-dimensional array type whose element type is BIT or BOOLEAN.

For the binary operators **and**, **or**, **nand**, **nor**, **xor**, and **xnor**, the operands shall both be of the same base type, or one operand shall be of a scalar type and the other operand shall be a one-dimensional array whose element type is the scalar type. The result type is the same as the base type of the operands if both operands are scalars of the same base type or both operands are arrays, or the same as the base type of the array operand if one operand is a scalar and the other operand is an array.

If both operands are one-dimensional arrays, the operands shall be arrays of the same length, the operation is performed on matching elements of the arrays, and the result is an array with the same index range as the left operand. If one operand is a scalar and the other operand is a one-dimensional array, the operation is performed on the scalar operand with each element of the array operand. The result is an array with the same index range as the array operand.

For the unary operator **not**, the result type is the same as the base type of the operand. If the operand is a one-dimensional array, the operation is performed on each element of the operand, and the result is an array with the same index range as the operand.

The effects of the logical operators are defined in the following tables. The symbol T represents TRUE for type BOOLEAN, '1' for type BIT; the symbol F represents FALSE for type BOOLEAN, '0' for type BIT.

A	B	A **and** B
T	T	T
T	F	F
F	T	F
F	F	F

A	B	A **or** B
T	T	T
T	F	T
F	T	T
F	F	F

A	B	A **xor** B
T	T	F
T	F	T
F	T	T
F	F	F

A	B	A **nand** B
T	T	F
T	F	T
F	T	T
F	F	T

A	B	A **nor** B
T	T	F
T	F	F
F	T	F
F	F	T

A	B	A **xnor** B
T	T	T
T	F	F
F	T	F
F	F	T

A	**not** A
T	F
F	T

For the short-circuit operations **and**, **or**, **nand**, and **nor** on types BIT and BOOLEAN, the right operand is evaluated only if the value of the left operand is not sufficient to determine the result of the operation. For operations **and** and **nand**, the right operand is evaluated only if the value of the left operand is T; for operations **or** and **nor**, the right operand is evaluated only if the value of the left operand is F.

The unary logical operators **and**, **or**, **nand**, **nor**, **xor**, and **xnor** are referred to as logical reduction operators. The logical reduction operators are predefined for any one-dimensional array type whose element type is BIT or BOOLEAN. The result type for the logical reduction operators is the same as the element type of the operand.

The values returned by the logical reduction operators are defined as follows. In the remainder of this subclause, the values of their arguments are referred to as R.

— The **and** operator returns a value that is the logical and of the elements of R. That is, if R is a null array, the return value is '1' if the element type of R is BIT or TRUE if the element type of R is BOOLEAN. Otherwise, the return value is the result of a binary **and** operation. The left argument of the binary **and** operation is the leftmost element of R. The right argument of the binary **and** operation is the result of a unary **and** operation with the argument being the rightmost (R'LENGTH – 1) elements of R.

— The **or** operator returns a value that is the logical or of the elements of R. That is, if R is a null array, the return value is '0' if the element type of R is BIT or FALSE if the element type of R is BOOLEAN. Otherwise, the return value is the result of a binary **or** operation. The left argument of the binary **or** operation is the leftmost element of R. The right argument of the binary **or** operation is the result of a unary **or** operation with the argument being the rightmost (R'LENGTH – 1) elements of R.

— The **xor** operator returns a value that is the logical exclusive-or of the elements of R. That is, if R is a null array, the return value is '0' if the element type of R is BIT or FALSE if the element type of R is BOOLEAN. Otherwise, the return value is the result of a binary **xor** operation. The left argument of the binary **xor** operation is the leftmost element of R. The right argument of the binary **xor** operation is the result of a unary **xor** operation with the argument being the rightmost (R'LENGTH – 1) elements of R.

— The **nand** operator returns a value that is the negated logical and of the elements of R. That is, the return value is the result of a **not** operation. The argument of the **not** operation is the result of a unary **and** operation with the argument being R.

— The **nor** operator returns a value that is the negated logical or of the elements of R. That is, the return value is the result of a **not** operation. The argument of the **not** operation is the result of a unary **or** operation with the argument being R.

— The **xnor** operator returns a value that is the negated logical exclusive-or of the elements of R. That is, the return value is the result of a **not** operation. The argument of the **not** operation is the result of a unary **xor** operation with the argument being R.

NOTE—All of the binary logical operators belong to the class of operators with the lowest precedence. The unary logical operators belong to the class of operators with the highest precedence.

9.2.3 Relational operators

Relational operators include tests for equality, inequality, and ordering of operands. The operands of each relational operator shall be of the same type. The result type of each ordinary relational operator (=, /=, <, <=, >, and >=) is the predefined type BOOLEAN. The result type of each matching relational operator (?=, ?/=, ?<, ?<=, ?>, and ?>=) is the same as the type of the operands (for scalar operands) or the the element type of the operands (for array operands).

Operator	Operation	Operand type	Result type
=	Equality	Any type, other than a file type or a protected type	BOOLEAN
/=	Inequality	Any type, other than a file type or a protected type	BOOLEAN
< <= > >=	Ordering	Any scalar type or discrete array type	BOOLEAN
?=	Matching equality	BIT or STD_ULOGIC	Same type
		Any one-dimensional array type whose element type is BIT or STD_ULOGIC	The element type
?/=	Matching inequality	BIT or STD_ULOGIC	Same type
		Any one-dimensional array type whose element type is BIT or STD_ULOGIC	The element type
?< ?<= ?> ?>=	Matching ordering	BIT or STD_ULOGIC	Same type

The equality and inequality operators (= and /=) are defined for all types other than file types and protected types. The equality operator returns the value TRUE if the two operands are equal and returns the value FALSE otherwise. The inequality operator returns the value FALSE if the two operands are equal and returns the value TRUE otherwise.

Two scalar values of the same type are equal if and only if the values are the same. Two composite values of the same type are equal if and only if for each element of the left operand there is a *matching element* of the right operand and vice versa, and the values of matching elements are equal, as given by the predefined equality operator for the element type. In particular, two null arrays of the same type are always equal. Two values of an access type are equal if and only if they both designate the same object or they both are equal to the null value for the access type.

For two record values, matching elements are those that have the same element identifier. For two one-dimensional array values, matching elements are those (if any) whose index values match in the following sense: the left bounds of the index ranges are defined to match; if two elements match, the elements immediately to their right are also defined to match. For two multidimensional array values, matching elements are those whose indices match in successive positions.

The ordinary ordering operators are defined for any scalar type and for any discrete array type. A *discrete array* is a one-dimensional array whose elements are of a discrete type. Each operator returns TRUE if the corresponding relation is satisfied; otherwise, the operator returns FALSE.

For scalar types, ordering is defined in terms of the relative values. For discrete array types, the relation < (less than) is defined such that the left operand is less than the right operand if and only if the left operand is a null array and the right operand is a non-null array.

Otherwise, both operands are non-null arrays, and one of the following conditions is satisfied:

a) The leftmost element of the left operand is less than that of the right, or

b) The leftmost element of the left operand is equal to that of the right, and the tail of the left operand is less than that of the right (the tail consists of the remaining elements to the right of the leftmost element and can be null).

The relation <= (less than or equal) for discrete array types is defined to be the inclusive disjunction of the results of the < and = operators for the same two operands. The relations > (greater than) and >= (greater than or equal) are defined to be the complements of the <= and < operators, respectively, for the same two operands.

The matching relational operators are predefined for the predefined type BIT and for the type STD_ULOGIC defined in package STD_LOGIC_1164. For operands of type BIT, each matching relational operator returns '1' if the corresponding ordinary relational operator applied to the operands returns TRUE, and returns the value '0' otherwise.

For the matching ordering operators applied to operands of type STD_ULOGIC, if either operand is the value '–', an error is reported in a manner equivalent to execution of the following assertion statement (see 10.3):

```
assert FALSE
   report "STD_LOGIC_1164: '-' operand for matching ordering operator"
   severity ERROR;
```

For operands of type STD_ULOGIC, the value returned by the matching equality operator is defined in the following table:

?=	**Right operand**								
Left operand	**'U'**	**'X'**	**'0'**	**'1'**	**'Z'**	**'W'**	**'L'**	**'H'**	**'–'**
'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'1'
'X'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'1'
'0'	'U'	'X'	'1'	'0'	'X'	'X'	'1'	'0'	'1'
'1'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'1'
'Z'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'1'
'W'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'1'
'L'	'U'	'X'	'1'	'0'	'X'	'X'	'1'	'0'	'1'
'H'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'1'
'–'	'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'

For operands of type STD_ULOGIC, the value returned by the matching ordering operator ?< is defined in the following table:

?<	**Right operand**								
Left operand	**'U'**	**'X'**	**'0'**	**'1'**	**'Z'**	**'W'**	**'L'**	**'H'**	**'–'**
'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'X'
'X'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'0'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'1'	'U'	'X'	'0'	'0'	'X'	'X'	'0'	'0'	'X'
'Z'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'W'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'L'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'H'	'U'	'X'	'0'	'0'	'X'	'X'	'0'	'0'	'X'
'–'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'

For operands of type STD_ULOGIC, the value returned by the matching inequality operator is defined to be the result of applying the **not** operator to the result of applying the ?= operator to the operands. The value returned by the matching ordering operator ?<= is defined to be the result of applying the binary **or** operator to the results of applications of the ?< and ?= operators to the operands. The value returned by the matching ordering operator ?> is the result of applying the **not** operator to the result of applying the ?<= operator to the operands. The value returned by the matching ordering operator ?>= is the result of applying the **not** operator to the result of applying the ?< operator to the operands. In each case, the **not** and **or** operators are those declared in the package IEEE.STD_LOGIC_1164.

The matching equality and matching inequality operators are also defined for any one-dimensional array type whose element type is BIT or STD_ULOGIC. The operands shall be arrays of the same length. The matching equality operator for the element type is applied to matching elements of the operands to form an intermediate array of type BIT_VECTOR (in the case of operands whose element type is BIT) or STD_ULOGIC_VECTOR (in the case of operands whose element type is STD_ULOGIC). The result of the matching equality operator applied to the operands is then the result of applying the unary **and** operator to the intermediate array. The result of the matching inequality operator is the result of applying the **not** operator to the result of applying the unary **and** operator to the intermediate array. In each case, the **not** and **and** operators are either the predefined operators or those declared in the package IEEE.STD_LOGIC_1164, as appropriate.

9.2.4 Shift operators

The shift operators **sll**, **srl**, **sla**, **sra**, **rol**, and **ror** are defined for any one-dimensional array type whose element type is either of the predefined types BIT or BOOLEAN.

Operator	Operation	Left operand type	Right operand type	Result type
sll	Shift left logical	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
srl	Shift right logical	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
sla	Shift left arithmetic	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
sra	Shift right arithmetic	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
rol	Rotate left logical	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
ror	Rotate right logical	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left

The index range of the return value of each shift operator is the same as the index range of the left operand.

The values returned by the shift operators are defined as follows. In the remainder of this subclause, the values of their leftmost arguments are referred to as L and the values of their rightmost arguments are referred to as R.

— The **sll** operator returns a value that is L logically shifted left by R index positions. That is, if R is 0 or if L is a null array, the return value is L. Otherwise, a basic shift operation replaces L with a value that is the result of a concatenation whose left argument is the rightmost (L'LENGTH – 1) elements of L and whose right argument is T'LEFT, where T is the element type of L. If R is positive, this basic shift operation is repeated R times to form the result. If R is negative, then the return value is the value of the expression L **srl** –R.

— The **srl** operator returns a value that is L logically shifted right by R index positions. That is, if R is 0 or if L is a null array, the return value is L. Otherwise, a basic shift operation replaces L with a value that is the result of a concatenation whose right argument is the leftmost (L'LENGTH – 1) elements of L and whose left argument is T'LEFT, where T is the element type of L. If R is positive, this basic shift operation is repeated R times to form the result. If R is negative, then the return value is the value of the expression L **sll** –R.

— The **sla** operator returns a value that is L arithmetically shifted left by R index positions. That is, if R is 0 or if L is a null array, the return value is L. Otherwise, a basic shift operation replaces L with a value that is the result of a concatenation whose left argument is the rightmost (L'LENGTH – 1) elements of L and whose right argument is L(L'RIGHT). If R is positive, this basic shift operation is repeated R times to form the result. If R is negative, then the return value is the value of the expression L **sra** –R.

— The **sra** operator returns a value that is L arithmetically shifted right by R index positions. That is, if R is 0 or if L is a null array, the return value is L. Otherwise, a basic shift operation replaces L with a value that is the result of a concatenation whose right argument is the leftmost (L'LENGTH – 1) elements of L and whose left argument is L(L'LEFT). If R is positive, this basic shift operation is repeated R times to form the result. If R is negative, then the return value is the value of the expression L **sla** –R.

— The **rol** operator returns a value that is L rotated left by R index positions. That is, if R is 0 or if L is a null array, the return value is L. Otherwise, a basic rotate operation replaces L with a value that is the result of a concatenation whose left argument is the rightmost (L'LENGTH – 1) elements of L and whose right argument is L(L'LEFT). If R is positive, this basic rotate operation is repeated R

times to form the result. If R is negative, then the return value is the value of the expression L **ror** –R.

— The **ror** operator returns a value that is L rotated right by R index positions. That is, if R is 0 or if L is a null array, the return value is L. Otherwise, a basic rotate operation replaces L with a value that is the result of a concatenation whose right argument is the leftmost (L'LENGTH – 1) elements of L and whose left argument is L(L'RIGHT). If R is positive, this basic rotate operation is repeated R times to form the result. If R is negative, then the return value is the value of the expression L **rol** –R.

NOTE 1—The logical operators may be overloaded, for example, to disallow negative integers as the second argument.

NOTE 2—The subtype of the result of a shift operator is the same as that of the left operand.

9.2.5 Adding operators

The adding operators + and – are predefined for any numeric type and have their conventional mathematical meaning. The concatenation operator & is predefined for any one-dimensional array type.

Operator	Operation	Left operand type	Right operand type	Result type
+	Addition	Any numeric type	Same type	Same type
–	Subtraction	Any numeric type	Same type	Same type
&	Concatenation	Any one-dimensional array type	Same array type	Same array type
		Any one-dimensional array type	The element type	Same array type
		The element type	Any one-dimensional array type	Same array type
		The element type	The element type	Any one-dimensional array type

For concatenation, there are three mutually exclusive cases, as follows:

a) If both operands are one-dimensional arrays of the same type, the result of the concatenation is a one-dimensional array of this same type whose length is the sum of the lengths of its operands, and whose elements consist of the elements of the left operand (in left-to-right order) followed by the elements of the right operand (in left-to-right order).

   If both operands are null arrays, then the result of the concatenation is the right operand. Otherwise, the direction and bounds of the result are determined as follows: Let S be the index subtype of the base type of the result. The direction of the result of the concatenation is the direction of S, and the left bound of the result is S'LEFT.

b) If one of the operands is a one-dimensional array and the type of the other operand is the element type of this aforementioned one-dimensional array, the result of the concatenation is given by the rules in case a), using in place of the other operand an implicit array having this operand as its only element. Both the left and right bounds of the index subtype of this implicit array is S'LEFT, and the direction of the index subtype of this implicit array is the direction of S, where S is the index subtype of the base type of the result.

c) If both operands are of the same type and it is the element type of some one-dimensional array type, the type of the result is this one-dimensional array type. In this case, each operand is treated as the one element of an implicit array, and the result of the concatenation is determined as in case a). The bounds and direction of the index subtypes of the implicit arrays are determined as in the case of the implicit array in case b).

In all cases, it is an error if either bound of the index range of the result does not belong to the index subtype of the type of the result, unless the result is a null array. It is also an error if any element of the result does not belong to the element subtype of the type of the result.

Examples:

```
subtype BYTE is BIT_VECTOR (7 downto 0);
type MEMORY is array (Natural range <>) of BYTE;

--  The following concatenation accepts two BIT_VECTORs and returns
--  a BIT_VECTOR [case a)]:

constant ZERO: BYTE := "0000" & "0000";

--  The next two examples show that the same expression can represent
--  either case a) or case c), depending on the context of
--  the expression.

--  The following concatenation accepts two BIT_VECTORS and returns
--  a BIT_VECTOR [case a)]:

constant C1: BIT_VECTOR := ZERO & ZERO;

--  The following concatenation accepts two BIT_VECTORs and returns
--  a MEMORY [case c)]:

constant C2: MEMORY := ZERO & ZERO;

--  The following concatenation accepts a BIT_VECTOR and a MEMORY,
--  returning a MEMORY [case b)]:

constant C3: MEMORY := ZERO & C2;

--  The following concatenation accepts a MEMORY and a BIT_VECTOR,
--  returning a MEMORY [case b)]:

constant C4: MEMORY := C2 & ZERO;

--  The following concatenation accepts two MEMORYs and returns
--  a MEMORY [case a)]:

constant C5: MEMORY := C2 & C3;

type R1 is range 0 to 7;
type R2 is range 7 downto 0;

type T1 is array (R1 range <>) of Bit;
type T2 is array (R2 range <>) of Bit;

subtype S1 is T1(R1);
subtype S2 is T2(R2);

constant K1: S1 := (others => '0');
constant K2: T1 := K1(1 to 3) & K1(3 to 4);  --  K2'Left = 0
```



```
                                                --      and K2'Right = 4
constant K3: T1 := K1(5 to 7) & K1(1 to 2);     --  K3'Left = 0
                                                --      and K3'Right = 4
constant K4: T1 := K1(2 to 1) & K1(1 to 2);     --  K4'Left = 0
                                                --      and K4'Right = 1

constant K5: S2 := (others => '0');
constant K6: T2 := K5(3 downto 1) & K5(4 downto 3);  --  K6'Left = 7
                                                --      and K6'Right = 3
constant K7: T2 := K5(7 downto 5) & K5(2 downto 1);  --  K7'Left = 7
                                                --      and K7'Right = 3
constant K8: T2 := K5(1 downto 2) & K5(2 downto 1);  --  K8'Left = 7
                                                --      and K8'Right = 6
```

NOTE 1—For a given concatenation whose operands are of the same type, there may be visible more than one array type that could be the result type according to the rules of case c). The concatenation is ambiguous and therefore an error if, using the overload resolution rules of 4.5 and 12.5, the type of the result is not uniquely determined.

NOTE 2—Additionally, for a given concatenation, there may be visible array types that allow both case a) and case c) to apply. The concatenation is again ambiguous and therefore an error if the overload resolution rules cannot be used to determine a result type uniquely.

9.2.6 Sign operators

Signs + and – are predefined for any numeric type and have their conventional mathematical meaning: they respectively represent the identity and negation functions. For each of these unary operators, the operand and the result have the same type.

Operator	Operation	Operand type	Result type
+	Identity	Any numeric type	Same type
–	Negation	Any numeric type	Same type

NOTE—Because of the relative precedence of signs + and – in the grammar for expressions, a signed operand shall not follow a multiplying operator, the exponentiating operator **, or the operators **abs** and **not**. For example, the syntax does not allow the following expressions:

```
A/+B          --  An illegal expression.
A**-B         --  An illegal expression.
```

However, these expressions may be rewritten legally as follows:

```
A/(+B)        --  A legal expression.
A ** (-B)     --  A legal expression.
```

9.2.7 Multiplying operators

The operators * and / are predefined for any integer and any floating-point type and have their conventional mathematical meaning; the operators **mod** and **rem** are predefined for any integer type. For each of these operators, the operands and the result are of the same type.

Operator	Operation	Left operand type	Right operand type	Result type
*	Multiplication	Any integer type	Same type	Same type
		Any floating-point type	Same type	Same type
/	Division	Any integer type	Same type	Same type
		Any floating-point type	Same type	Same type
mod	Modulus	Any integer type	Same type	Same type
rem	Remainder	Any integer type	Same type	Same type

Integer division and remainder are defined by the following relation:

A = (A/B) * B + (A **rem** B)

where (A **rem** B) has the sign of A and an absolute value less than the absolute value of B. Integer division satisfies the following identity:

(–A)/B = – (A/B) = A/(–B)

The result of the modulus operation is such that (A **mod** B) has the sign of B and an absolute value less than the absolute value of B; in addition, for some integer value N, this result shall satisfy the relation:

A = B * N + (A **mod** B)

In addition to the preceding table, the multiplying operators are predefined for any physical type.

Operator	Operation	Left operand type	Right operand type	Result type
*	Multiplication	Any physical type	INTEGER	Same as left
		Any physical type	REAL	Same as left
		INTEGER	Any physical type	Same as right
		REAL	Any physical type	Same as right
/	Division	Any physical type	INTEGER	Same as left
		Any physical type	REAL	Same as left
		Any physical type	The same type	*Universal integer*
mod	Modulus	Any physical type	Same type	Same type
rem	Remainder	Any physical type	Same type	Same type

Multiplication of a value P of a physical type T_p by a value I of type INTEGER is equivalent to the following computation:

T_p'Val(T_p'Pos(P) * I)

Multiplication of a value P of a physical type T_p by a value F of type REAL is equivalent to the following computation:

T_p'Val(INTEGER(REAL(T_p'Pos(P)) * F))

Division of a value P of a physical type T_p by a value I of type INTEGER is equivalent to the following computation:

T_p'Val(T_p'Pos(P) / I)

Division of a value P of a physical type T_p by a value F of type REAL is equivalent to the following computation:

T_p'Val(INTEGER(REAL(T_p'Pos(P)) / F))

Division of a value P of a physical type T_p by a value P2 of the same physical type is equivalent to the following computation:

T_p'Pos(P) / T_p'Pos(P2)

The computation of P **mod** P2, where P and P2 are values of a physical type T_p, is equivalent to the following computation:

T_p'Val(T_p'Pos(P) **mod** T_p'Pos(P2))

The computation of P **rem** P2, where P and P2 are values of a physical type T_p, is equivalent to the following computation:

T_p'Val(T_p'Pos(P) **rem** T_p'Pos(P2))

Examples:

```
  5      rem    3       =  2
  5      mod    3       =  2
(-5)     rem    3       = -2
(-5)     mod    3       =  1
(-5)     rem   (-3)     = -2
(-5)     mod   (-3)     = -2
  5      rem   (-3)     =  2
  5      mod   (-3)     = -1
  5 ns   rem    3 ns    = 2 ns
  5 ns   mod    3 ns    = 2 ns
(-5 ns)  rem    3 ns    = -2 ns
(-5 ns)  mod    3 ns    = 1 ns
  1 ns   mod    300 ps  = 100 ps
(-1 ns)  mod    300 ps  = 200 ps
```

NOTE—Because of the precedence rules (see 9.2.1), the expression "–5 **rem** 2" is interpreted as "–(5 **rem** 2)" and not as "(–5) **rem** 2."

9.2.8 Miscellaneous operators

The unary operator **abs** is predefined for any numeric type.

Operator	Operation	Operand type	Result type
abs	Absolute value	Any numeric type	Same numeric type

The *exponentiating* operator ** is predefined for each integer type and for each floating-point type. In either case the right operand, called the *exponent*, is of the predefined type INTEGER.

Operator	Operation	Left operand type	Right operand type	Result type
**	Exponentiation	Any integer type	INTEGER	Same as left
		Any floating-point type	INTEGER	Same as left

Exponentiation with an integer exponent is equivalent to repeated multiplication of the left operand by itself for a number of times indicated by the absolute value of the exponent and from left to right; if the exponent is negative, then the result is the reciprocal of that obtained with the absolute value of the exponent. Exponentiation with a negative exponent is only allowed for a left operand of a floating-point type. Exponentiation by a zero exponent results in the value one. Exponentiation of a value of a floating-point type is approximate.

9.2.9 Condition operator

The unary operator ?? is predefined for type BIT defined in package STANDARD (see 16.3).

Operator	Operation	Operand type	Result type
??	Condition conversion	BIT	BOOLEAN

Conversion of a value of type BIT converts '1' to TRUE and '0' to FALSE. The conversion operator may be overloaded for other types.

In certain circumstances, the condition operator is implicitly applied to an expression that occurs as a condition in any of the following places:

- After **until** in the condition clause of a wait statement (see 10.2)
- After **assert** in an assertion, either in an assertion statement (see 10.3) or in a concurrent assertion statement (see 11.5)
- After **if** or **elsif** in an if statement (see 10.8)
- After **while** in a while iteration scheme of a loop statement (see 10.10)
- After **when** in a next statement (see 10.11)
- After **when** in an exit statement (see 10.12)
- After **when** in a conditional signal assignment statement (see 10.5.3), either in a signal assignment statement or in a concurrent signal assignment statement
- After **when** in a conditional variable assignment statement (see 10.6.3)
- After **if** or **elsif** in an if generate statement (see 11.8)
- In a guard condition in a block statement (see 11.2)
- In a Boolean expression in a PSL declaration or a PSL directive

The condition operator implicitly applied, if any, is either the predefined operator for type BIT or an overloaded operator, determined as follows. If, without overload resolution (see 12.5), the expression is of type BOOLEAN defined in package STANDARD, or if, assuming a rule requiring the expression to be of type BOOLEAN defined in package STANDARD, overload resolution can determine at least one interpretation of each constituent of the innermost complete context including the expression, then the condition operator is not applied. Otherwise, the condition operator is implicitly applied, and the type of the expression with the implicit application shall be BOOLEAN defined in package STANDARD.

Example:

```
use IEEE.STD_LOGIC_1164.all;
signal S: STD_ULOGIC;

assert S;  -- implicit conversion applied
```

NOTE 1—The condition operator is not implicitly applied if there is at least one interpretation of the expression as being of type BOOLEAN. If overload resolution yields more than one such interpretation, the expression is of type BOOLEAN but ambiguous. In cases where the condition operator is implicitly applied to the expression, overload resolution may yield multiple interpretations, in which case the expression is ambiguous. The expression is only legal if there is exactly one interpretation of type BOOLEAN without the condition operator, or failing that, one interpretation of type BOOLEAN with the condition operator.

NOTE 2—The condition operator is defined for type STD_ULOGIC defined in package STD_LOGIC_1164 (see 16.7). Conversion of a value of type STD_ULOGIC converts '1' and 'H' to TRUE and all other values to FALSE.

9.3 Operands

9.3.1 General

The operands in an expression include names (that denote objects, values, or attributes that result in a value), literals, aggregates, function calls, qualified expressions, type conversions, and allocators. In addition, an expression enclosed in parentheses may be an operand in an expression. Names are defined in 8.1; the other kinds of operands are defined in 9.3.2 through 9.3.7.

9.3.2 Literals

A literal is either a numeric literal, an enumeration literal, a string literal, a bit string literal, or the literal **null**.

```
literal ::=
      numeric_literal
    | enumeration_literal
    | string_literal
    | bit_string_literal
    | null

numeric_literal ::=
      abstract_literal
    | physical_literal
```

Numeric literals include literals of the abstract types *universal_integer* and *universal_real*, as well as literals of physical types. Abstract literals are defined in 15.5; physical literals are defined in 5.2.4.1.

Enumeration literals are literals of enumeration types. They include both identifiers and character literals. Enumeration literals are defined in 5.2.2.1.

String and bit string literals are representations of one-dimensional arrays of characters. The type of a string or bit string literal shall be determinable solely from the context in which the literal appears, excluding the literal itself but using the fact that the type of the literal shall be a one-dimensional array of a character type. The lexical structure of string and bit string literals is defined in Clause 15.

For a non-null array value represented by either a string or bit string literal, the direction and bounds of the index range of the array value are determined according to the rules for positional array aggregates, where the number of elements in the aggregate is equal to the length (see 15.7 and 15.8) of the string or bit string literal. For a null array value represented by either a string or bit string literal, the direction and leftmost bound of the index range of the array value are determined as follows: the direction and nominal leftmost bound of the index range of the array value are determined as in the non-null case. If there is a value to the left of the nominal leftmost bound (given by the 'LEFTOF attribute), then the leftmost bound is the nominal leftmost bound, and the rightmost bound is the value to the left of the nominal leftmost bound. Otherwise, the leftmost bound is the value to the right of the nominal leftmost bound, and the rightmost bound is the nominal leftmost bound.

For a null array value represented by either a string or bit string literal, it is an error if the base type of the index subtype of the array type does not have at least two values.

The character literals corresponding to the graphic characters contained within a string literal or a bit string literal shall be visible at the place of the string literal.

The literal **null** represents the null access value for any access type.

Evaluation of a literal yields the corresponding value.

Examples:

```
3.14159_26536     --  A literal of type universal_real.
5280              --  A literal of type universal_integer.
10.7 ns           --  A literal of a physical type.
O"4777"           --  A bit string literal.
"54LS281"         --  A string literal.
""                --  A string literal representing a null array.
```

9.3.3 Aggregates

9.3.3.1 General

An aggregate is a basic operation (see 5.1) that combines one or more values into a composite value of a record or array type.

```
aggregate ::=
    ( element_association { , element_association } )

element_association ::=
    [ choices => ] expression

choices ::= choice { | choice }

choice ::=
      simple_expression
    | discrete_range
    | element_simple_name
```


| **others**

Each element association associates an expression with elements (possibly none). An element association is said to be *named* if the elements are specified explicitly by choices; otherwise, it is said to be *positional*. For a positional association, each element is implicitly specified by position in the textual order of the elements in the corresponding type declaration.

Both named and positional associations can be used in the same aggregate, with all positional associations appearing first (in textual order) and all named associations appearing next (in any order, except that it is an error if any associations follow an **others** association). Aggregates containing a single element association shall always be specified using named association in order to distinguish them from parenthesized expressions.

An element association with a choice that is an element simple name is only allowed in a record aggregate. An element association with a choice that is a simple expression or a discrete range is only allowed in an array aggregate: a simple expression specifies the element at the corresponding index value, whereas a discrete range specifies the elements at each of the index values in the range. Except as described in 9.3.3.3, the discrete range, and, in particular, the direction specified or implied by the discrete range, has no significance other than to define the set of choices implied by the discrete range. An element association with the choice **others** is allowed in either an array aggregate or a record aggregate if the association appears last and has this single choice; it specifies all remaining elements, if any.

Each element of the value defined by an aggregate shall be represented once and only once in the aggregate.

The type of an aggregate shall be determinable solely from the context in which the aggregate appears, excluding the aggregate itself but using the fact that the type of the aggregate shall be a composite type. The type of an aggregate in turn determines the required type for each of its elements.

9.3.3.2 Record aggregates

If the type of an aggregate is a record type, the element names given as choices shall denote elements of that record type. If the choice **others** is given as a choice of a record aggregate, it shall represent at least one element. An element association with more than one choice, or with the choice **others**, is only allowed if the elements specified are all of the same type. The expression of an element association shall have the type of the associated record elements.

A record aggregate is evaluated as follows. The expressions given in the element associations are evaluated in an order (or lack thereof) not defined by the language. The expression of a named association is evaluated once for each associated element. A check is made that the value of each element of the aggregate belongs to the subtype of this element. It is an error if this check fails.

9.3.3.3 Array aggregates

For an aggregate of a one-dimensional array type, each choice shall specify values of the index type, and the expression of each element association shall be of either the element type or the type of the aggregate. If the type of the expression of an element association is the type of the aggregate, then either the element association shall be positional or the choice shall be a discrete range.

For an element association with a choice that is a discrete range and an expression of the element type of the aggregate, the value of the expression is the element at each index value in the range.

For an element association with a choice that is a discrete range and an expression of the type of the aggregate, each element of the value of the expression is the value of the element of the aggregate at the matching index value in the range. The *matching index value* for an element of the value of the expression is

determined as follows: the leftmost element of the value matches the left bound of the range; if an element matches an index value, the element immediately to its right matches the index value immediately to the right in the range. It is an error if the length of the discrete range differs from the length of the value of the expression.

For a positional association with an expression of the element type of the aggregate, the expression specifies one element of the aggregate value. For a positional association with an expression of the type of the aggregate, the expression specifies a number of matching elements (see 9.2.3) of the aggregate value given by the length of the value of the expression.

An aggregate of an *n*-dimensional array type, where *n* is greater than 1, is written as a one-dimensional aggregate in which the index subtype of the aggregate is given by the first index position of the array type, and the expression specified for each element association is an (*n*–1)-dimensional array or array aggregate, which is called a *subaggregate*. A string or bit string literal is allowed as a subaggregate in the place of any aggregate of a one-dimensional array of a character type.

Apart from a final element association with the single choice **others**, the rest (if any) of the element associations of an array aggregate shall be either all positional or all named. A named association of an array aggregate is allowed to have a choice that is not locally static, or likewise a choice that is a null range, only if the aggregate includes a single element association and this element association has a single choice. An **others** choice is locally static if the applicable index constraint is locally static.

The index range of an array aggregate that has an **others** choice shall be determinable from the context. That is, an array aggregate with an **others** choice shall appear only in one of the following contexts:

a) As an actual associated with a formal parameter, formal generic, or formal port (or member thereof), where either the formal (or the member) is declared to be of a fully constrained array subtype, or the formal designator is a slice name

b) As the default expression defining the default initial value of a port declared to be of a fully constrained array subtype

c) As the default expression for a generic constant declared to be of a fully constrained array subtype

d) As the result expression of a function, where the corresponding function result type is a fully constrained array subtype

e) As a value expression in an assignment statement, where the target is a declared object (or member thereof), and either the subtype of the target is a fully constrained array subtype or the target is a slice name

f) As the expression defining the initial value of a constant or variable object, where that object is declared to be of a fully constrained array subtype

g) As the expression defining the default values of signals in a signal declaration, where the corresponding subtype is a fully constrained array subtype

h) As the expression defining the value of an attribute in an attribute specification, where that attribute is declared to be of a fully constrained array subtype

i) As the operand of a qualified expression whose type mark denotes a fully constrained array subtype

j) As a choice in a case statement whose expression is of a one-dimensional character array type and is one of the following:

   — The name of an object whose subtype is locally static, in which case the index range of the aggregate is the index range of the subtype of the object

   — An indexed name whose prefix is one of the members of this list and whose indexing expressions are locally static expressions, in which case the index range of the aggregate is the index range of the element subtype of the prefix

— A slice name whose prefix is one of the members of this list and whose discrete range is a locally static discrete range, in which case the index range of the aggregate is the discrete range

— A function call whose return type mark denotes a locally static subtype, in which case the index range of the aggregate is the index range of the subtype denoted by the return type mark

— A qualified expression or type conversion whose type mark denotes a locally static subtype, in which case the index range of the aggregate is the index range of the subtype denoted by the type mark

— An expression described in this list and enclosed in parentheses, in which case the index range of the aggregate is the index range of the subtype defined for the enclosed expression

In each case, the applicable index constraint is locally static.

k) As a subaggregate nested within an aggregate, where that aggregate itself appears in one of these contexts

The direction of the index range of an array that does not have an **others** choice are determined as follows:

— If the aggregate appears in one of the contexts in the preceding list, then the direction of the index range of the aggregate is that of the corresponding fully constrained array subtype, or that of the range of the corresponding slice name, as appropriate.

— If the aggregate does not appear in one of the contexts in the preceding list and an element association in the aggregate has a choice that is a discrete range and an expression that is of the type of the aggregate, then the direction of the index range of the aggregate is that of the discrete range.

— Otherwise, the direction of the index range of the aggregate is that of the index subtype of the base type of the aggregate.

The bounds of an array that does not have an **others** choice are determined as follows. For an aggregate that has named associations, the leftmost and rightmost bounds are determined by the direction of the index range of the aggregate and the smallest and largest choices given. For a positional aggregate, the leftmost bound is determined by the applicable index constraint if the aggregate appears in one of the contexts in the preceding list; otherwise, the leftmost bound is given by S'LEFT where S is the index subtype of the base type of the array. In either case, the rightmost bound is determined by the direction of the index range and the number of elements.

It is an error if the direction of the index range of an aggregate is determined by the context, and an element association has a choice that is a discrete range and an expression that is of the type of the aggregate, and the direction of the discrete range differs from that of the index range of the aggregate. If an aggregate has a given element association with a choice that is a discrete range and an expression that is of the type of the aggregate, then it is an error if any other element association has a choice that is a discrete range whose direction differs from that of the choice of the given element association.

The evaluation of an array aggregate that is not a subaggregate proceeds in two steps. First, the choices of this aggregate and of its subaggregates, if any, are evaluated in some order (or lack thereof) that is not defined by the language. Second, the expressions of the element associations of the array aggregate are evaluated in some order that is not defined by the language; the expression of a named association in which the expression is of the element type of the aggregate is evaluated once for each associated element. The evaluation of a subaggregate consists of this second step (the first step is omitted since the choices have already been evaluated).

For the evaluation of an aggregate that is not a null array, a check is made that the index values defined by choices belong to the corresponding index subtypes, and also that the value of each element of the aggregate belongs to the subtype of this element. For a multidimensional aggregate of dimension n, a check is made that all (n–1)-dimensional subaggregates have the same bounds. It is an error if any one of these checks fails.

9.3.4 Function calls

A function call invokes the execution of a function body. The call specifies the name of the function to be invoked and specifies the actual parameters, if any, to be associated with the formal parameters of the function. Execution of the function body results in a value of the type declared to be the result type in the declaration of the invoked function.

function_call ::=
    *function*_name [(actual_parameter_part)]

actual_parameter_part ::= *parameter*_association_list

For each formal parameter of a function, a function call shall specify exactly one corresponding actual parameter. This actual parameter is specified either explicitly, by an association element (other than the actual part **open**) in the association list, or in the absence of such an association element, by a default expression (see 6.5.2).

It is an error if the function name denotes an uninstantiated function.

Evaluation of a function call includes evaluation of the actual parameter expressions specified in the call and evaluation of the default expressions associated with formal parameters of the function that do not have actual parameters associated with them. In both cases, the resulting value shall belong to the subtype of the associated formal parameter. (If the formal parameter is of an unconstrained or partially constrained composite type, then any undefined index ranges of subelements of the formal parameter are determined as described in 5.3.2.2.) The function body is executed using the actual parameter values and default expression values as the values of the corresponding formal parameters.

NOTE 1—If a name (including one used as a prefix) has an interpretation both as a function call and an indexed name, then the innermost complete context is used to disambiguate the name. If, after applying this rule, there is not exactly one interpretation of the name, then the name is ambiguous. See 12.5.

NOTE 2—A call to a formal generic function uses the parameter names and default expressions defined in the formal generic function declaration and the parameter subtypes and result subtype of the associated actual generic function.

9.3.5 Qualified expressions

A qualified expression is a basic operation (see 5.1) that is used to explicitly state the type, and possibly the subtype, of an operand that is an expression or an aggregate.

qualified_expression ::=
      type_mark ' (expression)
    | type_mark ' aggregate

The operand shall have the same type as the base type of the type mark. The value of a qualified expression is the value of the operand. The evaluation of a qualified expression evaluates the operand and converts it to the subtype denoted by the type mark.

NOTE—Whenever the type of an enumeration literal or aggregate is not known from the context, a qualified expression can be used to state the type explicitly.

9.3.6 Type conversions

A type conversion provides for explicit conversion between closely related types.

type_conversion ::= type_mark (expression)

The target type of a type conversion is the base type of the type mark, and the target subtype of a type conversion is the type or subtype denoted by the type mark. The type of the operand of a type conversion shall be determined by applying the rules of 12.5 to the operand considered as a complete context. (In particular, the type of the operand must be determinable independent of the target type). Furthermore, the operand of a type conversion is not allowed to be the literal **null**, an allocator, an aggregate, or a string literal. An expression enclosed by parentheses is allowed as the operand of a type conversion only if the expression alone is allowed.

If the type mark denotes a subtype, conversion consists of conversion to the target type followed by a check that the result of the conversion belongs to the subtype.

In certain cases, an implicit subtype conversion is performed. A subtype conversion involves a type conversion in which the target subtype is the subtype to which the operand is converted and the target type is the base type of the target subtype.

Explicit type conversions are allowed between *closely related types*. In particular, a type is closely related to itself. Other types are closely related only under the following conditions:

— *Abstract numeric types*—Any abstract numeric type is closely related to any other abstract numeric type.

— *Array types*—Two array types are closely related if and only if the types have the same dimensionality and the element types are closely related

No other types are closely related.

In a type conversion where the target type is an abstract numeric type, the operand can be of any integer or floating-point type. The value of the operand is converted to the target type, which shall also be an integer or floating-point type. The conversion of a floating-point value to an integer type rounds to the nearest integer; if the value is halfway between two integers, rounding may be up or down.

In the case of conversions between numeric types, it is an error if the result of the conversion fails to satisfy a constraint imposed by the type mark.

In a type conversion where the target type is an array type, the following rules apply:

a) If the target subtype is an array type or subtype for which the index ranges are not defined, then, for each index position, the index range of the result is determined as follows:

   — If the index type of the operand and the index type of the target type are not closely related, then the direction and nominal left bound of the index range of the result are the direction and left bound, respectively, of the corresponding index subtype of the target type. For a non-null range, the left bound of the index range is the nominal left bound, and the right bound is determined by the number of values in the corresponding index range of the operand. For a null range, if there is a value to the left of the nominal left bound (given by the 'LEFTOF attribute), then the left bound is the nominal left bound, and the right bound is the value to the left of the nominal left bound; otherwise, the left bound is the value to the right of the nominal left bound, and the right bound is the nominal left bound. For either a non-null or a null range, it is an error if the base type of the corresponding index subtype of the target type does not include sufficient values for the index range of the result.

   — If the index type of the operand and the index type of the target type are closely related, then the bounds of the index range of the result are obtained by converting the bounds of the index range of the operand to the index type of the target type, and the direction of the index range of the result is the direction of the index type of the operand.

b) If the target subtype is an array subtype for which the index ranges are defined, then the bounds of the result are those imposed by the target subtype.

In either case, the value of each element of the result is that of the matching element of the operand (see 9.2.3) converted to the element subtype of the target subtype.

In the case of conversions between array types, if the target subtype is an array type for which the index ranges are not defined, then, for each index position, a check is made that the bounds of the result belong to the corresponding index subtype of the target type. If the target subtype is an array subtype for which the index ranges are defined, a check is made that for each element of the operand there is a matching element of the target subtype, and vice versa. It is an error if any of these checks fail.

In a subtype conversion where the target type is a record type, the value of each element of the result is that of the matching element of the operand (see 9.2.3) converted to the subtype of the element of the result.

In certain cases, an implicit type conversion will be performed. An implicit conversion of an operand of type *universal_integer* to another integer type, or of an operand of type *universal_real* to another floating-point type, can only be applied if the operand is either a numeric literal or an attribute, or if the operand is an expression consisting of the division of a value of a physical type by a value of the same type; such an operand is called a *convertible* universal operand. An implicit conversion of a convertible universal operand is applied if and only if the innermost complete context determines a unique (numeric) target type for the implicit conversion, and there is no legal interpretation of this context without this conversion.

NOTE 1—Two array types may be closely related even if corresponding index positions have different directions.

NOTE 2—Two distinct record types are not closely related, even if they have the same element identifiers and element subtypes. A record type is, however, closely related to itself. Hence, an operand of a record type can be converted to a subtype of the record type.

9.3.7 Allocators

The evaluation of an allocator creates an object and yields an access value that designates the object.

allocator ::=
    new subtype_indication
  | **new** qualified_expression

The type of the object created by an allocator is the base type of the type mark given in either the subtype indication or the qualified expression. For an allocator with a subtype indication, the initial value of the created object is the same as the default initial value for an explicitly declared variable of the designated subtype. For an allocator with a qualified expression, this expression defines the initial value of the created object.

The type of the access value returned by an allocator shall be determinable solely from the context, but using the fact that the value returned is of an access type having the named designated type.

The only allowed form of constraint in the subtype indication of an allocator is an array constraint or a record constraint. If an allocator includes a subtype indication and if the type of the object created is an array type or a record type, then the subtype indication shall denote a fully constrained subtype. A subtype indication that is part of an allocator shall not include a resolution indication.

If the type of the created object is an array type or a record type, then the created object is always fully constrained. If the allocator includes a subtype indication, the created object is constrained by the subtype. If the allocator includes a qualified expression, the created object is constrained by the bounds of the initial value defined by that expression. For other types, the subtype of the created object is the subtype defined by the subtype of the access type definition.

For the evaluation of an allocator, the elaboration of the subtype indication or the evaluation of the qualified expression is first performed. The new object is then created, and the object is then assigned its initial value. Finally, an access value that designates the created object is returned.

In the absence of explicit deallocation, an implementation shall guarantee that any object created by the evaluation of an allocator remains allocated for as long as this object or one of its subelements is accessible directly or indirectly; that is, as long as it can be denoted by some name.

NOTE 1—Procedure deallocate is implicitly declared for each access type. This procedure provides a mechanism for explicitly deallocating the storage occupied by an object created by an allocator.

NOTE 2—An implementation may (but need not) deallocate the storage occupied by an object created by an allocator, once this object has become inaccessible.

Examples:

```
new NODE                          -- Takes on default initial value.
new NODE'(15 ns, null)            -- Initial value is specified.
new NODE'(Delay => 5  ns,
          \Next\=> Stack)         -- Initial value is specified.
new BIT_VECTOR'("00110110")       -- Constrained by initial value.
new STRING (1 to 10)              -- Constrained by index constraint.
new STRING                        -- Illegal: must be constrained.
```

9.4 Static expressions

9.4.1 General

Certain expressions are said to be *static*. Similarly, certain discrete ranges are said to be static, and the type marks of certain subtypes are said to denote static subtypes.

There are two categories of static expression. Certain forms of expression can be evaluated during the analysis of the design unit in which they appear; such an expression is said to be *locally static*. Certain forms of expression can be evaluated as soon as the design hierarchy in which they appear is elaborated; such an expression is said to be *globally static*.

9.4.2 Locally static primaries

An expression is said to be locally static if and only if every operator in the expression denotes an implicitly defined operator or an operator defined in one of the packages STD_LOGIC_1164, NUMERIC_BIT, NUMERIC_STD, NUMERIC_BIT_UNSIGNED, or NUMERIC_STD_UNSIGNED in library IEEE, and if every primary in the expression is a *locally static primary*, where a locally static primary is defined to be one of the following:

a) A literal of any type other than type TIME

b) A constant (other than a deferred constant) explicitly declared by a constant declaration with a locally static subtype or with an unconstrained or partially constrained composite subtype for which the applicable constraints are locally static, and initialized with a locally static expression

c) A formal generic constant of a generic-mapped subprogram or package (whether explicitly declared or equivalent to a subprogram or package instance, respectively), declared with a locally static subtype and for which the associated actual is a locally static expression

d) An alias whose aliased name (given in the corresponding alias declaration) is a locally static primary and for which the subtype with which the aliased object is viewed is a locally static subtype

e) A function call whose function name denotes an implicitly defined operation or an operation defined in one of the packages STD_LOGIC_1164, NUMERIC_BIT, NUMERIC_STD,

NUMERIC_BIT_UNSIGNED, or NUMERIC_STD_UNSIGNED in library IEEE and whose actual parameters are each locally static expressions

f) A predefined attribute that is a value, other than the predefined attributes 'INSTANCE_NAME and 'PATH_NAME, and whose prefix is either a locally static subtype or is an object name that is of a locally static subtype

g) A predefined attribute that is a function, other than the predefined attribute 'VALUE with a prefix whose base type is the predefined type TIME, and other than the predefined attributes 'EVENT, 'ACTIVE, 'LAST_EVENT, 'LAST_ACTIVE, 'LAST_VALUE, 'DRIVING, and 'DRIVING_VALUE, whose prefix is either a locally static subtype or is an object that is of a locally static subtype, and whose actual parameter (if any) is a locally static expression

h) A user-defined attribute whose value is defined by a locally static expression

i) A qualified expression whose type mark denotes a locally static subtype and whose operand is a locally static expression

j) A type conversion whose type mark denotes a locally static subtype and whose expression is a locally static expression

k) A locally static expression enclosed in parentheses

l) An array aggregate in which all expressions in element associations are locally static expressions, all simple expressions in choices are locally static expressions, all discrete ranges in choices are locally static discrete ranges, and the **others** choice, if present, is locally static

m) A record aggregate in which all expressions in element associations are locally static expressions

n) An indexed name whose prefix is a locally static primary and whose index expressions are all locally static expressions

o) A slice name whose prefix is a locally static primary and whose discrete range is a locally static discrete range

p) A selected name whose prefix is a locally static primary

A locally static range is either a range of the second form (see 5.2.1) whose bounds are locally static expressions, or a range of the first form whose prefix denotes either a locally static subtype or an object that is of a locally static subtype. A locally static range constraint is a range constraint whose range is locally static. A locally static scalar subtype is either a scalar base type or a scalar subtype formed by imposing on a locally static subtype a locally static range constraint. A locally static discrete range is either a locally static subtype or a locally static range.

A locally static index constraint is an index constraint for which each index subtype of the corresponding array type is locally static and in which each discrete range is locally static. A locally static array constraint is an array constraint with a locally static index constraint and, if the array element constraint is present, a locally static array element constraint. A locally static array subtype is a fully constrained array subtype formed by imposing on an unconstrained array type a locally static array constraint. The unconstrained array type shall have a locally static index subtype for each index position and a locally static index subtype for each index position of each array subelement, if any. A locally static record constraint is a record constraint with a locally static constraint in each record element constraint. A locally static record subtype is a fully constrained record type whose elements are all of locally static subtypes, or a fully constrained record subtype formed by imposing on an unconstrained record type a locally static record constraint. The unconstrained record type shall have a locally static index subtype for each index position of each array subelement, if any. A locally static access subtype is a subtype denoting an access type. A locally static file subtype is a subtype denoting a file type. A locally static formal generic type is a formal generic type of an explicit block statement or of a generic-mapped subprogram or package (whether explicitly declared or equivalent to a subprogram or package instance, respectively) for which the associated actual is a locally static subtype.

A locally static subtype is either a locally static scalar subtype, a locally static array subtype, a locally static record subtype, a locally static access subtype, a locally static file subtype, or a locally static formal generic type.

9.4.3 Globally static primaries

An expression is said to be globally static if and only if every operator in the expression denotes a pure function and every primary in the expression is a *globally static primary*, where a globally static primary is a primary that, if it denotes an object or a function, does not denote a dynamically elaborated named entity (see 14.6) and is one of the following:

a) A literal of type TIME

b) A locally static primary

c) A generic constant declared with a globally static subtype

d) A generate parameter

e) A constant (including a deferred constant) explicitly declared by a constant declaration with a globally static subtype or with an unconstrained or partially constrained composite subtype for which the applicable constraints are globally static

f) An alias whose aliased name (given in the corresponding alias declaration) is a globally static primary

g) An array aggregate, if and only if

   1) All expressions in its element associations are globally static expressions, and

   2) All ranges in its element associations are globally static ranges

h) A record aggregate, if and only if all expressions in its element associations are globally static expressions

i) A function call whose function name denotes a pure function and whose actual parameters are each globally static expressions

j) A predefined attribute that is one of 'SIMPLE_NAME, 'INSTANCE_NAME, or 'PATH_NAME

k) A predefined attribute that is a value, other than the predefined attributes 'SIMPLE_NAME, 'INSTANCE_NAME, and 'PATH_NAME, whose prefix is appropriate for a globally static attribute

l) A predefined attribute that is a function, other than the predefined attributes 'EVENT, 'ACTIVE, 'LAST_EVENT, 'LAST_ACTIVE, 'LAST_VALUE, 'DRIVING, and 'DRIVING_VALUE, whose prefix is appropriate for a globally static attribute, and whose actual parameter (if any) is a globally static expression

m) A user-defined attribute whose value is defined by a globally static expression

n) A qualified expression whose type mark denotes a globally static subtype and whose operand is a globally static expression

o) A type conversion whose type mark denotes a globally static subtype and whose expression is a globally static expression

p) An allocator of the first form (see 9.3.7) whose subtype indication denotes a globally static subtype

q) An allocator of the second form whose qualified expression is a globally static expression

r) A globally static expression enclosed in parentheses

s) A subelement or a slice of a globally static primary, provided that any index expressions are globally static expressions and any discrete ranges used in slice names are globally static discrete ranges

A prefix is *appropriate for a globally static attribute* if it denotes a signal, a constant, a type or subtype, a globally static function call, a variable that is not of an access type, or a variable of an access type whose designated subtype is fully constrained.

A globally static range is either a range of the second form (see 5.2.1) whose bounds are globally static expressions, or a range of the first form whose prefix is appropriate for a globally static attribute. A globally static range constraint is a range constraint whose range is globally static. A globally static scalar subtype is either a scalar base type or a scalar subtype formed by imposing on a globally static subtype a globally static range constraint. A globally static discrete range is either a globally static subtype or a globally static range.

A globally static index constraint is an index constraint for which each index subtype of the corresponding array type is globally static and in which each discrete range is globally static. A globally static array constraint is an array constraint with a globally static index constraint and, if the array element constraint is present, a globally static array element constraint. A globally static array subtype is a fully constrained array subtype formed by imposing on an unconstrained array type a globally static array constraint. A globally static record constraint is a record constraint with a globally static constraint in each record element constraint. A globally static record subtype is a fully constrained record type whose elements are all of globally static subtypes, or a fully constrained record subtype formed by imposing on an unconstrained record type a globally static record constraint. A globally static access subtype is a subtype denoting an access type. A globally static file subtype is a subtype denoting a file type. A globally static formal generic type is a formal generic type of a block statement (including an implied block statement representing a component instance or a bound design entity) or of a generic-mapped subprogram or package (whether explicitly declared or equivalent to a subprogram or package instance, respectively) for which the associated actual is a globally static subtype.

A globally static subtype is either a globally static scalar subtype, a globally static array subtype, a globally static record subtype, a globally static access subtype, a globally static file subtype, or a globally static formal generic type.

NOTE 1—An expression that is required to be a static expression shall either be a locally static expression or a globally static expression. Similarly, a range, a range constraint, a scalar subtype, a discrete range, an index constraint, an array constraint, an array subtype, a record constraint, or a record subtype that is required to be static shall either be locally static or globally static.

NOTE 2—The rules for globally static expressions imply that a declared constant or a generic may be initialized with an expression that is not globally static, for example, with a call to an impure function. The resulting constant value may be globally static, even though its initial value expression is not. Only interface constant, variable, and signal declarations require that their initial value expressions be static expressions.

9.5 Universal expressions

A *universal_expression* is either an expression that delivers a result of type *universal_integer* or one that delivers a result of type *universal_real*.

The same operations are predefined for the type *universal_integer* as for any integer type. The same operations are predefined for the type *universal_real* as for any floating-point type. In addition, these operations include the following multiplication and division operators:

Operator	Operation	Left operand type	Right operand type	Result type
*	Multiplication	*Universal real*	*Universal integer*	*Universal real*
		Universal integer	*Universal real*	*Universal real*
/	Division	*Universal real*	*Universal integer*	*Universal real*

The accuracy of the evaluation of a universal expression of type *universal_real* is at least as good as the accuracy of evaluation of expressions of the most precise predefined floating-point type supported by the implementation, apart from *universal_real* itself.

For the evaluation of an operation of a universal expression, the following rules apply. If the result is of type *universal_integer*, then the values of the operands and the result shall lie within the range of the integer type with the widest range provided by the implementation, excluding type *universal_integer* itself. If the result is of type *universal_real*, then the values of the operands and the result shall lie within the range of the floating-point type with the widest range provided by the implementation, excluding type *universal_real* itself.

NOTE—The predefined operators for the universal types are declared in package STANDARD as shown in 16.3.

10. Sequential statements

10.1 General

The various forms of sequential statements are described in this clause. Sequential statements are used to define algorithms for the execution of a subprogram or process; they execute in the order in which they appear.

```
sequence_of_statements ::=
    { sequential_statement }

sequential_statement ::=
      wait_statement
    | assertion_statement
    | report_statement
    | signal_assignment_statement
    | variable_assignment_statement
    | procedure_call_statement
    | if_statement
    | case_statement
    | loop_statement
    | next_statement
    | exit_statement
    | return_statement
    | null_statement
```

All sequential statements may be labeled. Such labels are implicitly declared at the beginning of the declarative part of the innermost enclosing process statement or subprogram body.

10.2 Wait statement

The wait statement causes the suspension of a process statement or a procedure.

```
wait_statement ::=
    [ label : ] wait [ sensitivity_clause ] [ condition_clause ] [ timeout_clause ] ;

sensitivity_clause ::= on sensitivity_list

sensitivity_list ::= signal_name { , signal_name }

condition_clause ::= until condition

condition ::= expression

timeout_clause ::= for time_expression
```

The sensitivity clause defines the *sensitivity set* of the wait statement, which is the set of signals to which the wait statement is sensitive. Each signal name in the sensitivity list identifies a given signal as a member of the sensitivity set. Each signal name in the sensitivity list shall be a static signal name, and each name shall denote a signal for which reading is permitted. If no sensitivity clause appears, the sensitivity set is constructed according to the following (recursive) rule:

The sensitivity set is initially empty. For each primary in the condition of the condition clause, if the primary is

- A simple name that denotes a signal, add the longest static prefix of the name to the sensitivity set.
- An expanded name that denotes a signal, add the longest static prefix of the name to the sensitivity set.
- A selected name whose prefix denotes a signal, add the longest static prefix of the name to the sensitivity set.
- An indexed name whose prefix denotes a signal, add the longest static prefix of the name to the sensitivity set and apply this rule to all expressions in the indexed name.
- A slice name whose prefix denotes a signal, add the longest static prefix of the name to the sensitivity set and apply this rule to any expressions appearing in the discrete range of the slice name.
- An attribute name, if the designator denotes a signal attribute, add the longest static prefix of the name of the implicit signal denoted by the attribute name to the sensitivity set; otherwise, apply this rule to the prefix of the attribute name.
- An aggregate, apply this rule to every expression appearing after the choices and the =>, if any, in every element association.
- A function call, apply this rule to every actual designator in every parameter association.
- An actual designator of **open** in a parameter association, do not add to the sensitivity set.
- A qualified expression, apply this rule to the expression or aggregate qualified by the type mark, as appropriate.
- A type conversion, apply this rule to the expression type converted by the type mark.
- A parenthesized expression, apply this rule to the expression enclosed within the parentheses.
- Otherwise, do not add to the sensitivity set.

This rule is also used to construct the sensitivity sets of the wait statements in the equivalent process statements for concurrent procedure call statements (11.4), concurrent assertion statements (11.5), and concurrent signal assignment statements (11.6). Furthermore, this rule is used to construct the sensitivity list of an implicit wait statement in a process statement whose process sensitivity list is the reserved word **all** (11.3).

If a signal name that denotes a signal of a composite type appears in a sensitivity list, the effect is as if the name of each scalar subelement of that signal appears in the list.

The condition clause specifies a condition that shall be met for the process to continue execution. If no condition clause appears, the condition clause **until** TRUE is assumed.

The timeout clause specifies the maximum amount of time the process will remain suspended at this wait statement. If no timeout clause appears, the timeout clause **for** (STD.STANDARD.TIME'HIGH – STD.STANDARD.NOW) is assumed. It is an error if the time expression in the timeout clause evaluates to a negative value.

The execution of a wait statement causes the time expression to be evaluated to determine the *timeout interval*. It also causes the execution of the corresponding process statement to be suspended, where the corresponding process statement is the one that either contains the wait statement or is the parent (see 4.3) of the procedure that contains the wait statement. The suspended process will resume, at the latest, immediately after the timeout interval has expired.

The suspended process also resumes as a result of an event occurring on any signal in the sensitivity set of the wait statement. If such an event occurs, the condition in the condition clause is evaluated. If the value of

the condition is FALSE, the process suspends again. Such repeated suspension does not involve the recalculation of the timeout interval.

It is an error if a wait statement appears in a function subprogram or in a procedure that has a parent that is a function subprogram. Furthermore, it is an error if a wait statement appears in an explicit process statement that includes a sensitivity list or in a procedure that has a parent that is such a process statement. Finally, it is an error if a wait statement appears within any subprogram whose body is declared within a protected type body, or within any subprogram that has a parent whose body is declared within a protected type body, or within any subprogram that has a parent whose body is declared within a protected type body.

Example:

```
type Arr is array (1 to 5) of BOOLEAN;
function F (P: BOOLEAN) return BOOLEAN;
signal S: Arr;
signal l, r: INTEGER range 1 to 5;

--  The following two wait statements have the same meaning:

wait until F(S(3)) and (S(l) or S(r));
wait on S(3), S, l, r until F(S(3)) and (S(l) or S(r));
```

NOTE 1—The wait statement **wait until** Clk = '1'; has semantics identical to

```
loop
   wait on Clk;
   exit when Clk = '1';
end loop;
```

because of the rules for the construction of the default sensitivity clause. These same rules imply that **wait until** TRUE; has semantics identical to **wait**;.

NOTE 2—The conditions that cause a wait statement to resume execution of its enclosing process may no longer hold at the time the process resumes execution if the enclosing process is a postponed process.

NOTE 3—The rule for the construction of the default sensitivity set implies that if a function call appears in a condition clause and the called function is an impure function, then any signals that are accessed by the function but that are not passed through the association list of the call are not added to the default sensitivity set for the condition by virtue of the appearance of the function call in the condition.

10.3 Assertion statement

An assertion statement checks that a specified condition is true and reports an error if it is not.

assertion_statement ::= [label :] assertion ;

assertion ::=
    assert condition
        [**report** expression]
        [**severity** expression]

If the **report** clause is present, it shall include an expression of predefined type STRING that specifies a message to be reported. If the **severity** clause is present, it shall specify an expression of predefined type SEVERITY_LEVEL that specifies the severity level of the assertion.

The **report** clause specifies a message string to be included in error messages generated by the assertion. In the absence of a **report** clause for a given assertion, the string “Assertion violation.” is the default value for

the message string. The **severity** clause specifies a severity level associated with the assertion. In the absence of a **severity** clause for a given assertion, the default value of the severity level is ERROR.

Execution of an assertion statement consists of evaluation of the Boolean expression specifying the condition. If the expression results in the value FALSE, then an *assertion violation* is said to occur. When an assertion violation occurs, the **report** and **severity** clause expressions of the corresponding assertion, if present, are evaluated. The specified message string and severity level (or the corresponding default values, if not specified) are then used to construct an error message.

The error message consists of at least the following:

a) An indication that this message is from an assertion
b) The value of the severity level
c) The value of the message string
d) The name of the design unit (see 13.1) containing the assertion

A line feed (LF) format effector occurring as an element of the message string is interpreted by the implementation as signifying the end of a line. The implementation shall transform the LF into the implementation-defined representation of the end of a line.

An implementation should continue execution of a model after occurrence of an assertion violation in which the severity level is NOTE, WARNING, or ERROR.

NOTE 1—An implementation may choose whether or not to continue execution of a model after occurrence of assertion violations with various severity levels. It may also give tool users ability to control simulator actions for assertions of various severity levels via mechanisms not specified by this standard.

NOTE 2—The inadvertent insertion of a semicolon between the condition and the reserved word **report** in an assertion statement does not cause an error. Rather, it causes the statement to be parsed as an assertion statement with no **report** or **severity** clause, followed by a report statement.

10.4 Report statement

A report statement displays a message.

```
report_statement ::=
    [ label : ]
        report expression
            [ severity expression ] ;
```

The **report** statement expression shall be of the predefined type STRING. The string value of this expression is included in the message generated by the report statement. If the **severity** clause is present, it shall specify an expression of predefined type SEVERITY_LEVEL. The severity clause specifies a severity level associated with the report. In the absence of a **severity** clause for a given report, the default value of the severity level is NOTE.

Execution of a report statement consists of the evaluation of the report expression and severity clause expression, if present. The specified message string and severity level (or corresponding default, if the severity level is not specified) are then used to construct a report message.

The report message consists of at least the following:

a) An indication that this message is from a report statement
b) The value of the severity level
c) The value of the message string
d) The name of the design unit containing the report statement

An LF format effector occurring as an element of the message string is interpreted by the implementation as signifying the end of a line. The implementation shall transform the LF into the implementation-defined representation of the end of a line.

An implementation should continue execution of a model after displaying a report message in which the severity level is NOTE, WARNING, or ERROR.

NOTE—An implementation may choose whether or not to continue execution of a model after execution of report statements with various severity levels. It may also give tool users ability to control simulator actions for report statements of various severity levels via mechanisms not specified by this standard.

Example:

```
report "Entering process P";
   -- A report statement with default severity NOTE.

report "Setup or Hold violation; outputs driven to 'X'"
   severity WARNING;
   -- Another report statement; severity is specified.
```

10.5 Signal assignment statement

10.5.1 General

A signal assignment statement modifies the projected output waveforms contained in the drivers of one or more signals (see 14.7.2), schedules a force for one or more signals, or schedules release of one or more signals (see 14.7.3).

signal_assignment_statement ::=
    [label :] simple_signal_assignment
  | [label :] conditional_signal_assignment
  | [label :] selected_signal_assignment

10.5.2 Simple signal assignments

10.5.2.1 General

simple_signal_assignment ::=
    simple_waveform_assignment
  | simple_force_assignment
  | simple_release_assignment

simple_waveform_assignment ::=
  target <= [delay_mechanism] waveform ;

simple_force_assignment ::=
  target <= **force** [force_mode] expression ;

simple_release_assignment ::=
  target <= **release** [force_mode] ;

force_mode ::= **in** | **out**

delay_mechanism ::=

transport
| [**reject** *time*_expression] **inertial**

target ::=
name
| aggregate

waveform ::=
waveform_element { , waveform_element }
| **unaffected**

If the target of the signal assignment statement is a name, then the name shall denote a signal. For a simple waveform assignment, the base type of the value component of each transaction produced by a waveform element on the right-hand side shall be the same as the base type of the signal denoted by the target. This form of signal assignment assigns right-hand side values to the drivers associated with a single (scalar or composite) signal. For a simple force assignment, the base type of the expression on the right-hand side shall be the same as the base type of the signal denoted by the target. This form of signal assignment schedules either a *driving-value force* or an *effective-value force* for a single signal, with the expression value being the *driving force value* or *effective force value*, respectively. A simple release assignment schedules a *driving-value release* or an *effective-value release* for a single signal.

If the target of the signal assignment statement is in the form of an aggregate, then the type of the aggregate shall be determinable from the context, excluding the aggregate itself but including the fact that the type of the aggregate shall be a composite type. Furthermore, the expression in each element association of the aggregate shall be a locally static name that denotes a signal. For a simple waveform assignment, the base type of the value component of each transaction produced by a waveform element on the right-hand side shall be the same as the base type of the aggregate. This form of signal assignment assigns slices or subelements of the right-hand side values to the drivers associated with the signal named as the corresponding slice or subelement of the aggregate. It is an error if the target of a simple force assignment or a simple release assignment is in the form of an aggregate.

If the target of a signal assignment statement is in the form of an aggregate, and if the expression in an element association of that aggregate is a signal name that denotes a given signal, then the given signal and each subelement thereof (if any) are said to be *identified* by that element association as targets of the assignment statement. It is an error if a given signal or any subelement thereof is identified as a target by more than one element association in such an aggregate. Furthermore, it is an error if an element association in such an aggregate contains an **others** choice, or if the element association contains a choice that is a discrete range and an expression of a type other than the aggregate type.

The right-hand side of a simple waveform assignment may optionally specify a delay mechanism. A delay mechanism consisting of the reserved word **transport** specifies that the delay associated with the first waveform element is to be construed as *transport* delay. Transport delay is characteristic of hardware devices (such as transmission lines) that exhibit nearly infinite frequency response: any pulse is transmitted, no matter how short its duration. If no delay mechanism is present, or if a delay mechanism including the reserved word **inertial** is present, the delay is construed to be *inertial* delay. Inertial delay is characteristic of switching circuits: a pulse whose duration is shorter than the switching time of the circuit will not be transmitted, or in the case that a pulse rejection limit is specified, a pulse whose duration is shorter than that limit will not be transmitted.

Every inertially delayed signal assignment has a *pulse rejection limit*. If the delay mechanism specifies inertial delay, and if the reserved word **reject** followed by a time expression is present, then the time expression specifies the pulse rejection limit. In all other cases, the pulse rejection limit is specified by the time expression associated with the first waveform element.

It is an error if the pulse rejection limit for any inertially delayed signal assignment statement is either negative or greater than the time expression associated with the first waveform element.

A simple signal assignment of the form

target <= [delay_mechanism] **unaffected** ;

has the same effect as replacing the given assignment with a null statement (not an assignment with a null waveform element).

The right-hand side of a simple force assignment or a simple release assignment may optionally specify a force mode. A force mode consisting of the reserved word **in** specifies that an effective-value force or an effective-value release is to be scheduled, and a force mode consisting of the reserved word **out** specifies that a driving-value force or a driving-value release is to be scheduled.

If the right-hand side of a simple force assignment or a simple release assignment does not specify a force mode, then a default force mode is used, as follows:

— If the target is a port or signal parameter of mode **in**, a force mode of **in** is used.
— If the target is a port of mode **out**, **inout**, or **buffer**, or a signal parameter of mode **out** or **inout**, a force mode of **out** is used.
— If the target is not a port or a signal parameter, a force mode of **in** is used.

It is an error if a force mode of **out** is specified and the target is a port of mode **in**.

It is an error if a simple force assignment schedules a driving value force or an effective value force for a member of a resolved composite signal.

NOTE 1—For a signal assignment whose target is a name, no subelement of the target can be of a protected type (see 5.3.1).

NOTE 2—For a signal assignment whose target is in the form of an aggregate, no element of the target can be of a protected type, nor can any subelement of any element of the target be of a protected type (see 5.3.1).

NOTE 3—If a right-hand side value expression is either a numeric literal or an attribute that yields a result of type *universal_integer* or *universal_real*, then an implicit type conversion is performed.

Examples:

```
-- Assignments using inertial delay:

-- The following three assignments are equivalent to each other:

Output_pin <= Input_pin after 10 ns;
Output_pin <= inertial Input_pin after 10 ns;
Output_pin <= reject 10 ns inertial Input_pin after 10 ns;

-- Assignments with a pulse rejection limit less than the time
expression:

Output_pin <= reject 5 ns inertial Input_pin after 10 ns;
Output_pin <= reject 5 ns inertial Input_pin after 10 ns,
                                   not Input_pin after 20 ns;

-- Assignments using transport delay:
```

```
Output_pin <= transport Input_pin after 10 ns;
Output_pin <= transport Input_pin after 10 ns,
                        not Input_pin after 20 ns;

-- Their equivalent assignments:

Output_pin <= reject 0 ns inertial Input_pin after 10 ns;
Output_pin <= reject 0 ns inertial Input_pin after 10 ns,
                                   not Input_pin after 20 ns;
```

10.5.2.2 Executing a simple assignment statement

The effect of execution of a simple waveform assignment statement is defined in terms of its effect upon the projected output waveforms (see 14.7.2) representing the current and future values of drivers of signals.

waveform_element ::=
    *value*_expression [**after** *time*_expression]
  | **null** [**after** *time*_expression]

The future behavior of the driver(s) for a given target is defined by transactions produced by the evaluation of waveform elements in the waveform of a simple waveform assignment statement. The first form of waveform element is used to specify that the driver is to assign a particular value to the target at the specified time. The second form of waveform element is used to specify that the driver of the signal is to be turned off, so that it (at least temporarily) stops contributing to the value of the target. This form of waveform element is called a *null waveform element*. It is an error if the target of a simple waveform assignment statement containing a null waveform element is not a guarded signal or an aggregate of guarded signals.

The base type of the time expression in each waveform element shall be the predefined physical type TIME as defined in package STANDARD. If the **after** clause of a waveform element is not present, then an implicit "**after** 0 ns" is assumed. It is an error if the time expression in a waveform element evaluates to a negative value.

Evaluation of a waveform element produces a single transaction. The time component of the transaction is determined by the current time added to the value of the time expression in the waveform element. For the first form of waveform element, the value component of the transaction is determined by the value expression in the waveform element. For the second form of waveform element, the value component is not defined by the language, but it is defined to be of the type of the target. A transaction produced by the evaluation of the second form of waveform element is called a *null transaction*.

For the execution of a simple waveform assignment statement whose target is of a scalar type, the waveform on its right-hand side is first evaluated. Evaluation of a waveform consists of the evaluation of each waveform element in the waveform. Thus, the evaluation of a waveform results in a sequence of transactions, where each transaction corresponds to one waveform element in the waveform. These transactions are called *new* transactions. It is an error if the sequence of new transactions is not in ascending order with respect to time. It is also an error if the value of any value expression in the waveform does not belong to the subtype of the target.

The sequence of transactions is then used to update the projected output waveform representing the current and future values of the driver associated with the simple waveform assignment statement. Updating a projected output waveform consists of the deletion of zero or more previously computed transactions (called *old* transactions) from the projected output waveform and the addition of the new transactions, as follows:

a) All old transactions that are projected to occur at or after the time at which the earliest new transaction is projected to occur are deleted from the projected output waveform.

b) The new transactions are then appended to the projected output waveform in the order of their projected occurrence.

If the initial delay is inertial delay according to the definitions of 10.5.2.1, the projected output waveform is further modified as follows:

1) All of the new transactions are marked.
2) An old transaction is marked if the time at which it is projected to occur is less than the time at which the first new transaction is projected to occur minus the pulse rejection limit.
3) For each remaining unmarked, old transaction, the old transaction is marked if it immediately precedes a marked transaction and its value component is the same as that of the marked transaction.
4) The transaction that determines the current value of the driver is marked.
5) All unmarked transactions (all of which are old transactions) are deleted from the projected output waveform.

For the purposes of marking transactions, any two successive null transactions in a projected output waveform are considered to have the same value component.

The execution of a simple waveform assignment statement whose target is of a composite type proceeds in a similar fashion, except that the evaluation of the waveform results in one sequence of transactions for each scalar subelement of the type of the target. Each such sequence consists of transactions whose value portions are determined by the values of the same scalar subelement of the value expressions in the waveform, and whose time portion is determined by the time expression corresponding to that value expression. Each such sequence is then used to update the projected output waveform of the driver of the matching subelement of the target. This applies both to a target that is the name of a signal of a composite type and to a target that is in the form of an aggregate.

For the execution of a simple force assignment whose target is of a scalar type, the expression on its right-hand side is first evaluated. It is an error if the value of the expression does not belong to the subtype of the target. The value of the expression is then used to schedule a driving-value force or an effective-value force.

The execution of a simple force assignment whose target is of a composite type proceeds in a similar fashion, except that the evaluation of the expression results in one value for each scalar subelement of the type of the target. Each such value is then used to schedule a driving-value force or an effective-value force of the matching subelement of the target.

For the execution of a simple release assignment whose target is of a scalar type, a driving-value release or an effective-value release is scheduled for the target. The execution of a simple release assignment whose target is of a composite type proceeds in a similar fashion, except that a driving-value release or an effective-value release is scheduled for each scalar subelement of the target.

It is an error if the target of a simple force assignment or a simple release assignment is a member of a resolved composite signal.

If a given procedure is declared by a declarative item that is not contained within a process statement, and if a simple waveform assignment statement appears in that procedure, then the target of the simple waveform assignment shall be a formal parameter of the given procedure or of a parent of that procedure, or an aggregate of such formal parameters. Similarly, if a given procedure is declared by a declarative item that is not contained within a process statement, and if a signal is associated with an **inout** or **out** mode signal parameter in a subprogram call within that procedure, then the signal so associated shall be a formal parameter of the given procedure or of a parent of that procedure.

NOTE 1—These rules guarantee that the driver affected by a simple waveform assignment statement is always statically determinable if the simple waveform assignment appears within a given process (including the case in which it appears

within a procedure that is declared within the given process). In this case, the affected driver is the one defined by the process; otherwise, the simple waveform assignment shall appear within a procedure, and the affected driver is the one passed to the procedure along with a signal parameter of that procedure. Simple force assignments and simple release assignments, on the other hand, do not involve drivers. Hence, the target of such an assignment occurring in a procedure not contained with a process statement need not be a signal parameter of the procedure.

NOTE 2—Overloading the operator "=" has no effect on the updating of a projected output waveform.

NOTE 3—Consider a signal assignment statement of the form

T <= **reject** t_r **inertial** e_1 **after** t_1 { , e_i **after** t_i }

The following relations hold:

$$0\ \text{ns} \le t_r \le t_1$$

and

$$0\ \text{ns} \le t_i < t_{i+1}$$

Note that, if $t_r = 0$ ns, then the waveform editing is identical to that for transport-delayed assignment; and if $t_r = t_1$, the waveform is identical to that for the statement

T <= e_1 after t_1 { , e_i after t_i }

NOTE 4—Consider the following signal assignment in some process:

```
S <= reject 15 ns inertial 12 after 20 ns, 18 after 41 ns;
```

where S is a signal of some integer type.

Assume that at the time this signal assignment is executed, the driver of S in the process has the following contents (the first entry is the current driving value):

1	2	2	12	5	8
NOW	+3 ns	+12 ns	+13 ns	+20 ns	+42 ns

(The times given are relative to the current time.) The updating of the projected output waveform proceeds as follows:

— The driver is truncated at 20 ns. The driver now contains the following pending transactions:

1	2	2	12
NOW	+3 ns	+12 ns	+13 ns

— The new waveforms are added to the driver. The driver now contains the following pending transactions:

1	2	2	12	12	18
NOW	+3 ns	+12 ns	+13 ns	+20 ns	+41 ns

— All new transactions are marked, as well as those old transactions that occur at less than the time of the first new waveform (20 ns) less the rejection limit (15 ns). The driver now contains the following pending transactions (marked transactions are in bold type):

1	**2**	2	12	**12**	**18**
NOW	**+3 ns**	+12 ns	+13 ns	**+20 ns**	**+41 ns**

— Each remaining unmarked transaction is marked if it immediately precedes a marked transaction and has the same value as the marked transaction. The driver now contains the following pending transactions:

1	**2**	2	**12**	**12**	**18**
NOW	**+3 ns**	+12 ns	**+13 ns**	**+20 ns**	**+41 ns**

— The transaction that determines the current value of the driver is marked, and all unmarked transactions are then deleted. The final driver contents are then as follows, after clearing the markings:

1	2	12	12	18
NOW	+3 ns	+13 ns	+20 ns	+41 ns

10.5.3 Conditional signal assignments

The conditional signal assignment represents an equivalent if statement that assigns values to signals or that forces or releases signals.

```
conditional_signal_assignment ::=
      conditional_waveform_assignment
    | conditional_force_assignment

conditional_waveform_assignment ::=
    target  <=  [ delay_mechanism ] conditional_waveforms ;

conditional_waveforms ::=
    waveform when condition
    { else waveform when condition }
    [ else waveform ]

conditional_force_assignment ::=
    target <= force [ force_mode ] conditional_expressions ;

conditional_expressions ::=
    expression when condition
    { else expression when condition }
    [ else expression ]
```

The delay mechanism for a conditional waveform assignment statement is discussed in 10.5.2.1.

For a given conditional signal assignment, there is an equivalent sequential statement with the same meaning. If the conditional signal assignment is of the form

```
target <= delay_mechanism
          waveform1   when condition1   else
          waveform2   when condition2   else
             .
             .
             .
          waveformN-1 when conditionN-1 else
          waveformN   when conditionN;
```

then the equivalent sequential statement is of the form

```
if condition1 then
   target <= delay_mechanism waveform1;
elsif condition2 then
   target <= delay_mechanism waveform2;
      .
      .
      .
elsif conditionN-1 then
   target <= delay_mechanism waveformN-1;
elsif conditionN then
   target <= delay_mechanism waveformN;
end if;
```

If the conditional signal assignment is of the form

```
target <= delay_mechanism
          waveform1   when condition1   else
          waveform2   when condition2   else
             .
             .
             .
          waveformN-1 when conditionN-1 else
          waveformN;
```

then the equivalent sequential statement is of the form

```
if condition1 then
   target <= delay_mechanism waveform1;
elsif condition2 then
   target <= delay_mechanism waveform2;
      .
      .
      .
elsif conditionN-1 then
   target <= delay_mechanism waveformN-1;
else
   target <= delay_mechanism waveformN;
end if;
```

If the conditional signal assignment is of the form


```
target <= force
          expression1   when condition1   else
          expression2   when condition2   else
             .
             .
             .
          expressionN-1 when conditionN-1 else
          expressionN   when conditionN;
```

then the equivalent sequential statement is of the form

```
if condition1 then
   target <= force expression1;
elsif condition2 then
   target <= force expression2;
      .
      .
      .
elsif conditionN-1 then
   target <= force expressionN-1;
elsif conditionN then
   target <= force expressionN;
end if;
```

If the conditional signal assignment is of the form

```
target <= force
   expression1 when condition1 else
   expression2 when condition2 else
      .
      .
      .
   expressionN-1 when conditionN-1 else
   expressionN;
```

then the equivalent sequential statement is of the form

```
if condition1 then
   target <= force expression1;
elsif condition2 then
   target <= force expression2;
      .
      .
      .
elsif conditionN-1 then
   target <= force expressionN-1;
else
   target <= force expressionN;
end if;
```

The characteristics of the target, waveforms, expressions, and conditions in the conditional assignment statement shall be such that the equivalent sequential statement is a legal statement.

If a label appears on the signal assignment statement containing the conditional signal assignment, then the same label appears on the equivalent sequential statement. If a delay mechanism appears in a conditional waveform assignment, then the same delay mechanism appears in every simple waveform assignment statement in the equivalent sequential statement.

Example:

```
S <= unaffected when Input_pin = S'Driving_Value else
     Input_pin after Buffer_Delay;
```

10.5.4 Selected signal assignments

The selected signal assignment represents an equivalent case statement that assigns values to signals or that forces or releases signals.

selected_signal_assignment ::=
    selected_waveform_assignment
  | selected_force_assignment

selected_waveform_assignment ::=
  with expression **select** [?]
    target <= [delay_mechanism] selected_waveforms ;

selected_waveforms ::=
  { waveform **when** choices , }
  waveform **when** choices

selected_force_assignment ::=
  with expression **select** [?]
    target <= **force** [force_mode] selected_expressions ;

selected_expressions ::=
  { expression **when** choices , }
  expression **when** choices

The delay mechanism for a selected waveform assignment statement is discussed in 10.5.2.1.

For a given selected signal assignment, there is an equivalent sequential statement with the same meaning. If the selected signal assignment is of the form

```
with expression select
   target <= delay_mechanism waveform1   when choice_list1,
                             waveform2   when choice_list2,
                                .
                                .
                                .
                             waveformN-1 when choice_listN-1,
                             waveformN   when choice_listN;
```

then the equivalent sequential statement is of the form

```
case expression is
   when choice_list1 =>
      target <= delay_mechanism waveform1;
```



```
   when choice_list2 =>
      target <= delay_mechanism waveform2;
         .
         .
         .
   when choice_listN-1 =>
      target <= delay_mechanism waveformN-1;
   when choice_listN =>
      target <= delay_mechanism waveformN;
end case;
```

If the selected signal assignment is of the form

```
with expression select
   target <= force expression1   when choice_list1,
                   expression2   when choice_list2,
                      .
                      .
                      .
                   expressionN-1 when choice_listN-1,
                   expressionN   when choice_listN;
```

then the equivalent sequential statement is of the form

```
case expression is
   when choice_list1 =>
      target <= force expression1;
   when choice_list2 =>
      target <= force expression2;
         .
         .
         .
   when choice_listN-1 =>
      target <= force expressionN-1;
   when choice_listN =>
      target <= force expressionN;
end case;
```

If a selected signal assignment statement includes the question mark delimiter, then the equivalent sequential statement includes a question mark delimiter after both occurrences of the reserved word **case**; otherwise the equivalent sequential statement does not include the question mark delimiters.

The characteristics of the select expression, the target, the waveforms, the expressions, and the choices in the selected assignment statement shall be such that the equivalent sequential statement is a legal statement.

If a label appears on the signal assignment statement containing the selected signal assignment, then the same label appears on the equivalent sequential statement. If a delay mechanism appears in a selected waveform assignment, then the same delay mechanism appears in every simple waveform assignment statement in the equivalent sequential statement.

10.6 Variable assignment statement

10.6.1 General

A variable assignment statement replaces the current value of a variable with a new value specified by an expression. The named variable and the right-hand side expression shall be of the same type.

variable_assignment_statement ::=
    [label :] simple_variable_assignment
  | [label :] conditional_variable_assignment
  | [label :] selected_variable_assignment

10.6.2 Simple variable assignments

10.6.2.1 General

simple_variable_assignment ::=
  target := expression ;

If the target of the variable assignment statement is a name, then the name shall denote a variable, and the base type of the expression on the right-hand side shall be the same as the base type of the variable denoted by that name. It is an error if the type of the target is a protected type. This form of variable assignment assigns the right-hand side value to a single (scalar or composite) variable.

If the target of the variable assignment statement is in the form of an aggregate, then the type of the aggregate shall be determinable from the context, excluding the aggregate itself but including the fact that the type of the aggregate shall be a composite type. The base type of the expression on the right-hand side shall be the same as the base type of the aggregate. Furthermore, the expression in each element association of the aggregate shall be a locally static name that denotes a variable. This form of variable assignment assigns each subelement or slice of the right-hand side value to the variable named as the corresponding subelement or slice of the aggregate.

If the target of a variable assignment statement is in the form of an aggregate, and if the locally static name in an element association of that aggregate denotes a given variable or denotes another variable of which the given variable is a subelement or slice, then the element association is said to *identify* the given variable as a target of the assignment statement. It is an error if a given variable is identified as a target by more than one element association in such an aggregate. Furthermore, it is an error if an element association in such an aggregate contains an **others** choice, or if the element association contains a choice that is a discrete range and an expression of a type other than the aggregate type.

For the execution of a variable assignment whose target is a variable name, the variable name and the expression are first evaluated. A check is then made that the value of the expression belongs to the subtype of the variable, except in the case of a variable that is of a composite type (in which case the assignment involves a subtype conversion). Finally, each subelement of the variable that is not forced is updated with the corresponding subelement of the expression. A design is erroneous if it depends on the order of evaluation of the target and source expressions of an assignment statement.

The execution of a variable assignment whose target is in the form of an aggregate proceeds in a similar fashion, except that each of the names in the aggregate is evaluated, and a subtype check is performed for each subelement or slice of the right-hand side value that corresponds to one of the names in the aggregate. For each variable denoted by a name corresponding to a subelement or slice of the right-hand side value, each subelement of the variable that is not forced is updated with the corresponding subelement of the subelement or slice of the right-hand side value.

An error occurs if the aforementioned subtype checks fail.

NOTE 1—If the right-hand side is either a numeric literal or an attribute that yields a result of type universal integer or universal real, then an implicit type conversion is performed.

NOTE 2—For a variable assignment whose target is a name, no subelement of the target can be of a protected type (see 5.3.1).

NOTE 3—For a variable assignment whose target is in the form of an aggregate, no element of the target can be of a protected type, nor can any subelement of any element of the target be of a protected type (see 5.3.1).

NOTE 4—The value of a composite variable or of any element or slice of a composite variable is considered to have changed if any of the subelements of the variable, element, or slice changes value.

10.6.2.2 Composite variable assignments

If the target of an assignment statement is a name denoting a composite variable (including a slice), the value assigned to the target is implicitly converted to the subtype of the composite variable; the result of this subtype conversion becomes the new value of the composite variable.

This means that the new value of each element of the composite variable is specified by the matching element (see 9.2.3) in the corresponding composite value obtained by evaluation of the expression. The subtype conversion checks that for each element of the composite variable there is a matching element in the composite value, and vice versa. An error occurs if this check fails.

10.6.3 Conditional variable assignments

The conditional variable assignment represents an equivalent if statement that assigns values to variables.

conditional_variable_assignment ::=
    target := conditional_expressions ;

For a given conditional variable assignment, there is an equivalent sequential statement with the same meaning. If the conditional variable assignment is of the form

If the conditional variable assignment is of the form

```
target :=
   expression1   when condition1   else
   expression2   when condition2   else
      .
      .
      .
   expressionN-1 when conditionN-1 else
   expressionN   when conditionN;
```

then the equivalent sequential statement is of the form

```
if condition1 then
   target := expression1;
elsif condition2 then
   target := expression2;
      .
      .
      .
elsif conditionN-1 then
   target := expressionN-1;
```

```
elsif conditionN
   target := expressionN;
end if;
```

If the conditional variable assignment is of the form

```
target :=
   expression1   when condition1   else
   expression2   when condition2   else
      .
      .
      .
   expressionN-1 when conditionN-1 else
   expressionN;
```

then the equivalent sequential statement is of the form

```
if condition1 then
   target := expression1;
elsif condition2 then
   target := expression2;
      .
      .
      .
elsif conditionN-1 then
   target := expressionN-1;
else
   target := expressionN;
end if;
```

The characteristics of the expressions and conditions in the conditional assignment statement shall be such that the equivalent sequential statement is a legal statement.

If a label appears on the variable assignment statement containing the conditional variable assignment, then the same label appears on the equivalent sequential statement.

Example:

```
N := V1 when S = S1 else
     V2 when S = S2;
```

10.6.4 Selected variable assignments

The selected variable assignment represents an equivalent case statement that assigns values to variables.

selected_variable_assignment ::=
    with expression **select** [?]
        target := selected_expressions ;

For a given selected variable assignment, there is an equivalent sequential statement with the same meaning. If the selected variable assignment is of the form

```
with expression select
   target := expression1   when choice_list1,
```



```
            expression2   when choice_list2,
               .
               .
               .
            expressionN-1 when choice_listN-1,
            expressionN   when choice_listN;
```

then the equivalent sequential statement is of the form

```
case expression is
   when choice_list1 =>
      target := expression1;
   when choice_list2 =>
      target := expression2;
         .
         .
         .
   when choice_listN-1 =>
      target := expressionN-1;
   when choice_listN =>
      target := expressionN;
end case;
```

If a selected variable assignment statement includes the question mark delimiter, then the equivalent sequential statement includes a question mark delimiter after both occurrences of the reserved word case; otherwise the equivalent sequential statement does not include the question mark delimiters.

The characteristics of the select expression, the expressions, and the choices in the selected assignment statement shall be such that the equivalent sequential statement is a legal statement.

If a label appears on the variable assignment statement containing the selected variable assignment, then the same label appears on the equivalent sequential statement.

10.7 Procedure call statement

A procedure call invokes the execution of a procedure body.

procedure_call_statement ::= [label :] procedure_call ;

procedure_call ::= *procedure*_name [(actual_parameter_part)]

The procedure name specifies the procedure body to be invoked. It is an error if the procedure name denotes an uninstantiated procedure. The actual parameter part, if present, specifies the association of actual parameters with formal parameters of the procedure.

For each formal parameter of a procedure, a procedure call shall specify exactly one corresponding actual parameter. This actual parameter is specified either explicitly, by an association element (other than the actual **open**) in the association list or, in the absence of such an association element, by a default expression (see 6.5.2).

Execution of a procedure call includes evaluation of the actual parameter expressions specified in the call and evaluation of the default expressions associated with formal parameters of the procedure that do not have actual parameters associated with them. In both cases, the resulting value shall belong to the subtype of

the associated formal parameter. (If the formal parameter is of an unconstrained or partially constrained composite type, then any undefined index ranges of subelements of the formal parameter are determined as described in 5.3.2.2.) The procedure body is executed using the actual parameter values and default expression values as the values of the corresponding formal parameters.

NOTE—A call to a formal generic procedure uses the parameter names and default expressions defined in the formal generic procedure declaration, and the parameter subtypes of the associated actual generic procedure.

10.8 If statement

An if statement selects for execution one or none of the enclosed sequences of statements, depending on the value of one or more corresponding conditions.

```
if_statement ::=
    [ if_label : ]
        if condition then
            sequence_of_statements
        { elsif condition then
            sequence_of_statements }
        [ else
            sequence_of_statements ]
        end if [ if_label ] ;
```

If a label appears at the end of an if statement, it shall repeat the if label.

For the execution of an if statement, the condition specified after **if**, and any conditions specified after **elsif**, are evaluated in succession (treating a final **else** as **elsif** TRUE **then**) until one evaluates to TRUE or all conditions are evaluated and yield FALSE. If one condition evaluates to TRUE, then the corresponding sequence of statements is executed; otherwise, none of the sequences of statements is executed.

10.9 Case statement

A case statement selects for execution one of a number of alternative sequences of statements; the chosen alternative is defined by the value of an expression.

```
case_statement ::=
    [ case_label : ]
        case [ ? ] expression is
            case_statement_alternative
            { case_statement_alternative }
        end case [ ? ] [ case_label ] ;

case_statement_alternative ::=
    when choices =>
        sequence_of_statements
```

A case statement shall include the question mark delimiter either in both places, in which case the case statement is called a *matching case statement*, or in neither place, in which case the case statement is called an *ordinary case statement*.

The expression shall be of a discrete type or of a one-dimensional array type whose element base type is a character type. This type shall be determined by applying the rules of 12.5 to the expression considered as a complete context, using the rule that the expression shall be of a discrete type or a one-dimensional character

array type. (In particular, the type of the case expression must be determinable independent of the type of the case statement choices.) It is an error if the type of the expression in a matching case statement is other than BIT, STD_ULOGIC, or a one-dimensional array type whose element type is BIT or STD_ULOGIC. Each choice in a case statement alternative shall be of the same type as the expression; the list of choices specifies for which values of the expression the alternative is chosen.

For an ordinary case statement, or for a matching case statement in which the expression is of type BIT or an array type whose element type is BIT, if the expression is the name of an object whose subtype is locally static, whether a scalar type or an array type, then each value of the subtype shall be represented once and only once in the set of choices of the case statement, and no other value is allowed; this rule is likewise applied if the expression is a qualified expression or type conversion whose type mark denotes a locally static subtype, or if the expression is a call to a function whose return type mark denotes a locally static subtype, or if the expression is an expression described in this paragraph and enclosed in parentheses.

For a matching case statement in which the expression is of type STD_ULOGIC, or an array type whose element type is STD_ULOGIC, if the expression is the name of an object whose subtype is locally static, whether a scalar type or an array type, then each value of the subtype, other than the scalar value '–' or an array value containing '–' as an element, shall be represented once and only once in the set of choices of the case statement. A value is represented by a choice if application of the predefined matching equality operator to the value and the choice gives the result '1'. It is an error if a choice does not represent a value of the subtype other than the scalar value '–' or an array value containing '–' as an element. This rule is likewise applied if the expression is a qualified expression or type conversion whose type mark denotes a locally static subtype, or if the expression is a call to a function whose return type mark denotes a locally static subtype, or if the expression is an expression described in this paragraph and enclosed in parentheses.

If the expression is of a one-dimensional character array type and is not described by either of the preceding two paragraphs, then the values of all of the choices, except the **others** choice, if present, shall be of the same length. Moreover, for an ordinary case statement, or for a matching case statement in which the expression is of an array type whose element type is BIT, each value of the (base) type of the expression shall be represented once and only once in the set of choices, and no other value is allowed. For a matching case statement in which the expression is of an array type whose element type is STD_ULOGIC, each value of the (base) type of the expression, other than an array value containing '–' as an element, shall be represented (as defined in the preceding paragraph) once and only once in the set of choices of the case statement. It is an error if a choice does not represent a value of the (base) type of the expression other than an array value containing '–' as an element. In all cases, it is an error if the value of the expression is not of the same length as the values of the choices. If there is only one choice and that choice is **others**, then the value of the expression may be of any length.

For other forms of expression in an ordinary case statement or in a matching case statement in which the expression is of type BIT, each value of the (base) type of the expression shall be represented once and only once in the set of choices, and no other value is allowed. For other forms of expression in a matching case statement in which the expression is of type STD_ULOGIC, each value of the (base) type of the expression, other than the scalar value '–', shall be represented once and only once in the set of choices of the case statement. It is an error if a choice does not represent a value of the (base) type of the expression other than the scalar value '–'.

All simple expressions and discrete ranges given as choices in a case statement shall be locally static. A choice defined by a discrete range stands for all values in the corresponding range. The choice **others** is only allowed for the last alternative and as its only choice; it stands for all values (possibly none) not given in the choices of previous alternatives. An element simple name (see 9.3.3.1) is not allowed as a choice of a case statement alternative. For a matching case statement in which the expression is of type STD_ULOGIC, or an array type whose element type is STD_ULOGIC, it is an error if application of the predefined matching equality operator to the values of any two distinct choices other than the choice **others** gives the result '1'.

If a label appears at the end of a case statement, it shall repeat the case label.

The execution of a case statement consists of the evaluation of the expression followed by the execution of the chosen sequence of statements. A sequence of statements in a given ordinary case statement alternative is the chosen sequence of statements if and only if the expression “E = V” evaluates to TRUE, where “E” is the expression, “V” is the value of one of the choices of the given case statement alternative (if a choice is a discrete range, then this latter condition is fulfilled when V is an element of the discrete range), and the operator “=” in the expression is the predefined “=” operator on the base type of E. A sequence of statements in a given matching case statement alternative is the chosen sequence of statements if and only if the condition “E ?= V” evaluates to TRUE or '1', where “E” and “V” are similarly defined and the operator “?=” is the predefined “?=” operator on the base type of E.

For a matching case statement in which the expression is of type STD_ULOGIC, or an array type whose element type is STD_ULOGIC, it is an error if the value of the expression is the scalar value '–' or an array value containing '–' as an element.

NOTE 1—The execution of a case statement chooses one and only one alternative, since the choices are exhaustive and mutually exclusive. A qualified expression whose type mark denotes a locally static subtype can often be used as the expression of a case statement to limit the number of choices that need be explicitly specified.

NOTE 2—An **others** choice is required in a case statement if the type of the expression is the type *universal_integer* (for example, if the expression is an integer literal), since this is the only way to cover all values of the type *universal_integer*.

NOTE 3—Overloading the operator “=” has no effect on the semantics of ordinary case statement execution. Similarly, overloading the operator “?=” has no effect on the semantics of matching case statement execution.

NOTE 4—An **others** choice is generally required in a matching case statement in which the expression is of type STD_ULOGIC, or an array type whose element type is STD_ULOGIC, since explicit choice values cannot be written to represent metalogical values of the expression. (Application of the predefined matching equality operator with a metalogical operand value gives the result 'X'.) Such expression values, which shall nonetheless be represented by a choice, are represented by the **others** choice.

10.10 Loop statement

A loop statement includes a sequence of statements that is to be executed repeatedly, zero or more times.

```
loop_statement ::=
    [ loop_label : ]
        [ iteration_scheme ] loop
            sequence_of_statements
        end loop [ loop_label ] ;

iteration_scheme ::=
      while condition
    | for loop_parameter_specification

parameter_specification ::=
    identifier in discrete_range
```

If a label appears at the end of a loop statement, it shall repeat the label at the beginning of the loop statement.

Execution of a loop statement is complete when the loop is left as a consequence of the completion of the iteration scheme (see the following), if any, or the execution of a next statement, an exit statement, or a return statement.

A loop statement without an iteration scheme specifies repeated execution of the sequence of statements.

For a loop statement with a **while** iteration scheme, the condition is evaluated before each execution of the sequence of statements; if the value of the condition is TRUE, the sequence of statements is executed; if FALSE, the iteration scheme is said to be *complete* and the execution of the loop statement is complete.

For a loop statement with a **for** iteration scheme, the *loop parameter* specification is the declaration of the loop parameter with the given identifier. The loop parameter is an object whose type is the base type of the discrete range. Within the sequence of statements, the loop parameter is a constant. Hence, a loop parameter is not allowed as the target of an assignment statement. Similarly, the loop parameter shall not be given as an actual corresponding to a formal of mode **out** or **inout** in an association list.

For the execution of a loop with a **for** iteration scheme, the discrete range is first evaluated. If the discrete range is a null range, the iteration scheme is said to be *complete* and the execution of the loop statement is therefore complete; otherwise, the sequence of statements is executed once for each value of the discrete range (subject to the loop not being left as a consequence of the execution of a next statement, an exit statement, or a return statement), after which the iteration scheme is said to be *complete*. Prior to each such iteration, the corresponding value of the discrete range is assigned to the loop parameter. These values are assigned in left-to-right order.

NOTE—A loop may be left as the result of the execution of a next statement if the loop is nested inside of an outer loop and the next statement has a loop label that denotes the outer loop.

10.11 Next statement

A next statement is used to complete the execution of one of the iterations of an enclosing loop statement (called *loop* in the following text). The completion is conditional if the statement includes a condition.

next_statement ::=
    [label :] **next** [*loop*_label] [**when** condition] ;

A next statement with a loop label is only allowed within the labeled loop and applies to that loop; a next statement without a loop label is only allowed within a loop and applies only to the innermost enclosing loop (whether labeled or not).

For the execution of a next statement, the condition, if present, is first evaluated. The current iteration of the loop is terminated if the value of the condition is TRUE or if there is no condition.

10.12 Exit statement

An exit statement is used to complete the execution of an enclosing loop statement (called *loop* in the following text). The completion is conditional if the statement includes a condition.

exit_statement ::=
    [label :] **exit** [*loop*_label] [**when** condition] ;

An exit statement with a loop label is only allowed within the labeled loop and applies to that loop; an exit statement without a loop label is only allowed within a loop and applies only to the innermost enclosing loop (whether labeled or not).

For the execution of an exit statement, the condition, if present, is first evaluated. Exit from the loop then takes place if the value of the condition is TRUE or if there is no condition.

10.13 Return statement

A return statement is used to complete the execution of the innermost enclosing function or procedure body.

return_statement ::=
    [label :] **return** [expression] ;

A return statement is only allowed within the body of a function or procedure, and it applies to the innermost enclosing function or procedure.

A return statement appearing in a procedure body shall not have an expression. A return statement appearing in a function body shall have an expression.

The value of the expression defines the result returned by the function. The type of this expression shall be the base type of the type mark given after the reserved word **return** in the specification of the function. It is an error if execution of a function completes by any means other than the execution of a return statement.

For the execution of a return statement, the expression (if any) is first evaluated and converted to the result subtype. The execution of the return statement is thereby completed if the conversion succeeds; so also is the execution of the enclosing subprogram. An error occurs at the place of the return statement if the conversion fails.

NOTE—If the expression is either a numeric literal, or an attribute that yields a result of type *universal_integer* or *universal_real*, then an implicit conversion of the result is performed.

10.14 Null statement

A null statement performs no action.

null_statement ::=
    [label :] **null** ;

The execution of the null statement has no effect other than to pass on to the next statement.

NOTE—The null statement can be used to specify explicitly that no action is to be performed when certain conditions are true, although it is never mandatory for this (or any other) purpose. This is particularly useful in conjunction with the case statement, in which all possible values of the case expression shall be covered by choices; for certain choices, it may be that no action is required.

11. Concurrent statements

11.1 General

The various forms of concurrent statements are described in this clause. Concurrent statements are used to define interconnected blocks and processes that jointly describe the overall behavior or structure of a design. Concurrent statements execute asynchronously with respect to each other.

concurrent_statement ::=
    block_statement
  | process_statement
  | concurrent_procedure_call_statement
  | concurrent_assertion_statement
  | concurrent_signal_assignment_statement
  | component_instantiation_statement
  | generate_statement
  | *PSL*_PSL_Directive

The primary concurrent statements are the block statement, which groups together other concurrent statements, and the process statement, which represents a single independent sequential process. Additional concurrent statements provide convenient syntax for representing simple, commonly occurring forms of processes, as well as for representing structural decomposition and regular descriptions.

Within a given simulation cycle, an implementation may execute concurrent statements in parallel or in some order. The language does not define the order, if any, in which such statements will be executed. A description that depends upon a particular order of execution of concurrent statements is erroneous.

All concurrent statements may be labeled. Such labels are implicitly declared at the beginning of the declarative part of the innermost enclosing entity declaration, architecture body, block statement, or generate statement.

11.2 Block statement

A block statement defines an internal block representing a portion of a design. Blocks may be hierarchically nested to support design decomposition.

block_statement ::=
    *block*_label :
        block [(*guard*_condition)] [**is**]
            block_header
            block_declarative_part
        begin
            block_statement_part
        end block [*block*_label] ;

block_header ::=
    [generic_clause
    [generic_map_aspect ;]]
    [port_clause
    [port_map_aspect ;]]

block_declarative_part ::=

```
    { block_declarative_item }

block_statement_part ::=
    { concurrent_statement }
```

If a guard condition appears after the reserved word **block**, then a signal with the simple name GUARD of predefined type BOOLEAN is implicitly declared at the beginning of the declarative part of the block, and the guard condition defines the value of that signal at any given time (see 14.7.4). The type of the guard condition shall be type BOOLEAN. Signal GUARD may be used to control the operation of certain statements within the block (see 11.6).

The implicit signal GUARD shall not have a source.

If a block header appears in a block statement, it explicitly identifies certain values or signals that are to be imported from the enclosing environment into the block and associated with formal generics or ports. The generic and port clauses define the formal generics and formal ports of the block (see 6.5.6.2 and 6.5.6.3); the generic map and port map aspects define the association of actuals with those formals (see 6.5.7.2 and 6.5.7.3). Such actuals are evaluated in the context of the enclosing declarative region.

If a label appears at the end of a block statement, it shall repeat the block label.

NOTE 1—The value of signal GUARD is always defined within the scope of a given block, and it does not implicitly extend to design entities bound to components instantiated within the given block. However, the signal GUARD may be explicitly passed as an actual signal in a component instantiation in order to extend its value to lower-level components.

NOTE 2—An actual appearing in a port association list of a given block can never denote a formal port of the same block.

11.3 Process statement

A process statement defines an independent sequential process representing the behavior of some portion of the design.

```
process_statement ::=
    [ process_label : ]
        [ postponed ] process [ ( process_sensitivity_list ) ] [ is ]
            process_declarative_part
        begin
            process_statement_part
        end [ postponed ] process [ process_label ] ;

process_sensitivity_list ::= all | sensitivity_list

process_declarative_part ::=
    { process_declarative_item }

process_declarative_item ::=
      subprogram_declaration
    | subprogram_body
    | subprogram_instantiation_declaration
    | package_declaration
    | package_body
    | package instantiation_declaration
    | type_declaration
    | subtype_declaration
```



```
    | constant_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause
    | group_template_declaration
    | group_declaration

process_statement_part ::=
    { sequential_statement }
```

If the reserved word **postponed** precedes the initial reserved word **process**, the process statement defines a *postponed process*; otherwise, the process statement defines a *nonpostponed process*.

If a process sensitivity list appears following the reserved word **process**, then the process statement is assumed to contain an implicit wait statement as the last statement of the process statement part; this implicit wait statement is of the form

wait on sensitivity_list ;

where the sensitivity list is determined in one of two ways. If the process sensitivity list is specified as a sensitivity list, then the sensitivity list of the wait statement is that following the reserved word **process**. If the process sensitivity list is specified using the reserved word **all**, then the sensitivity list of the wait statement is constructed by taking the union of the sets constructed from each of the statements in the process by applying the following rules:

- For each assertion, report, next, exit, or return statement, apply the rule of 10.2 to each expression in the statement, and construct the union of the resulting sets.
- For each assignment statement, apply the rule of 10.2 to each expression occurring in the assignment, including any expressions occurring in the index names or slice names in the target, and construct the union of the resulting sets.
- For each if statement, apply the rule of 10.2 to each condition and apply this rule recursively to each sequence of statements within the if statement, and construct the union of the resulting sets.
- For each case statement, apply the rule of 10.2 to the expression and apply this rule recursively to each sequence of statements within the case statement, and construct the union of the resulting sets.
- For each loop statement, apply the rule of 10.2 to each expression in the iteration scheme, if present, and apply this rule recursively to the sequence of statements within the loop statement, and construct the union of the resulting sets.
- For each procedure call statement, apply the rule of 10.2 to each actual designator (other than **open**) associated with each formal parameter of mode **in** or **inout**, and construct the union of the resulting sets.

Moreover, for each subprogram for which the process is a parent (see 4.3), the sensitivity list includes members of the set constructed by applying the preceding rule to the statements of the subprogram, but excluding the members that denote formal signal parameters or members of formal signal parameters of the subprogram or any of its parents.

It is an error if a process statement with the reserved word **all** as its process sensitivity list is the parent of a subprogram declared in a design unit other than that containing the process statement, and the subprogram reads an explicitly declared signal that is not a formal signal parameter or member of a formal signal parameter of the subprogram or of any of its parents. Similarly, it is an error if such a subprogram reads an

implicit signal whose explicit ancestor is not a formal signal parameter or member of a formal parameter of the subprogram or of any of its parents.

It is an error if any name that does not denote a static signal name (see 8.1) for which reading is permitted appears in the sensitivity list of a process statement.

If a process sensitivity list appears following the reserved word **process** in a process statement, then the process statement shall not contain an explicit wait statement. Similarly, if such a process statement is a parent of a procedure, then it is an error if that procedure contains a wait statement.

If the reserved word **postponed** appears at the end of a process statement, the process shall be a postponed process. If a label appears at the end of a process statement, the label shall repeat the process label.

It is an error if a variable declaration in a process declarative part declares a shared variable.

The execution of a process statement consists of the repetitive execution of its sequence of statements. After the last statement in the sequence of statements of a process statement is executed, execution will immediately continue with the first statement in the sequence of statements.

A process statement is said to be a *passive process* if neither the process itself, nor any procedure of which the process is a parent, contains a signal assignment statement. It is an error if a process or a concurrent statement, other than a passive process or a concurrent statement equivalent to such a process, appears in the entity statement part of an entity declaration.

NOTE 1—The rules in 11.3 imply that a process that has an explicit sensitivity list always has exactly one (implicit) wait statement in it, and that wait statement appears at the end of the sequence of statements in the process statement part. Thus, a process with a sensitivity list always waits at the end of its statement part; any event on a signal named in the sensitivity list will cause such a process to execute from the beginning of its statement part down to the end, where it will wait again. Such a process executes once through at the beginning of simulation, suspending for the first time when it executes the implicit wait statement.

NOTE 2—The time at which a process executes after being resumed by a wait statement (see 10.2) differs depending on whether the process is postponed or nonpostponed. When a nonpostponed process is resumed, it executes in the current simulation cycle (see 14.7.5). When a postponed process is resumed, it does not execute until a simulation cycle occurs in which the next simulation cycle is not a delta cycle. In this way, a postponed process accesses the values of signals that are the "final" values at the current simulated time.

NOTE 3—The conditions that cause a process to resume execution may no longer hold at the time the process resumes execution if the process is a postponed process.

NOTE 4—In general, it is not possible to determine at analysis time whether a process with the reserved word **all** as its process sensitivity list is the parent of a subprogram declared in a separate design unit and whether the rules for such a subprogram are met.

11.4 Concurrent procedure call statements

A concurrent procedure call statement represents a process containing the corresponding sequential procedure call statement.

concurrent_procedure_call_statement ::=
    [label :] [**postponed**] procedure_call ;

For any concurrent procedure call statement, there is an equivalent process statement. The equivalent process statement is a postponed process if and only if the concurrent procedure call statement includes the reserved word **postponed**. The equivalent process statement has a label if and only if the concurrent procedure call statement has a label; if the equivalent process statement has a label, it is the same as that of the concurrent procedure call statement. The equivalent process statement also has no sensitivity list, an

empty declarative part, and a statement part that consists of a procedure call statement followed by a wait statement.

The procedure call statement consists of the same procedure name and actual parameter part that appear in the concurrent procedure call statement.

If there exists a name that denotes a signal in the actual part of any association element in the concurrent procedure call statement, and that actual is associated with a formal parameter of mode **in** or **inout**, then the equivalent process statement includes a final wait statement with a sensitivity clause that is constructed by taking the union of the sets constructed by applying the rule of 10.2 to each actual part associated with a formal parameter.

Execution of a concurrent procedure call statement is equivalent to execution of the equivalent process statement.

Example:

```
CheckTiming (tPLH, tPHL, Clk, D, Q);  --  A concurrent procedure call
                                      --  statement.

process                               --  The equivalent process.
begin
   CheckTiming (tPLH, tPHL, Clk, D, Q);
   wait on Clk, D, Q;
end process;
```

NOTE 1—Concurrent procedure call statements make it possible to declare procedures representing commonly used processes and to create such processes easily by merely calling the procedure as a concurrent statement. The wait statement at the end of the statement part of the equivalent process statement allows a procedure to be called without having it loop interminably, even if the procedure is not necessarily intended for use as a process (i.e., it contains no wait statement). Such a procedure may persist over time (and thus the values of its variables retain state over time) if its outermost statement is a loop statement and the loop contains a wait statement. Similarly, such a procedure may be guaranteed to execute only once, at the beginning of simulation, if its last statement is a wait statement that has no sensitivity clause, condition clause, or timeout clause.

NOTE 2—The value of an implicitly declared signal GUARD has no effect on evaluation of a concurrent procedure call unless it is explicitly referenced in one of the actual parts of the actual parameter part of the concurrent procedure call statement.

11.5 Concurrent assertion statements

A concurrent assertion statement represents a passive process statement containing the specified assertion statement.

concurrent_assertion_statement ::=
    [label :] [**postponed**] assertion ;

For any concurrent assertion statement, there is an equivalent process statement. The equivalent process statement is a postponed process if and only if the concurrent assertion statement includes the reserved word **postponed**. The equivalent process statement has a label if and only if the concurrent assertion statement has a label; if the equivalent process statement has a label, it is the same as that of the concurrent assertion statement. The equivalent process statement also has no sensitivity list, an empty declarative part, and a statement part that consists of an assertion statement followed by a wait statement.

The assertion statement consists of the same condition, **report** clause, and **severity** clause that appear in the concurrent assertion statement.

If there exists a name that denotes a signal in the Boolean expression that defines the condition of the assertion, then the equivalent process statement includes a final wait statement with a sensitivity clause that is constructed by applying the rule of 10.2 to that expression; otherwise, the equivalent process statement contains a final wait statement that has no explicit sensitivity clause, condition clause, or timeout clause.

Execution of a concurrent assertion statement is equivalent to execution of the equivalent process statement.

If a concurrent statement is ambiguous and can be interpreted either as a concurrent assertion statement or as a PSL assertion directive, then it is interpreted as a concurrent assertion statement.

NOTE 1—Since a concurrent assertion statement represents a passive process statement, such a process has no outputs. Therefore, the execution of a concurrent assertion statement will never cause an event to occur. However, if the assertion is false, then the specified error message will be sent to the simulation report.

NOTE 2—The value of an implicitly declared signal GUARD has no effect on evaluation of the assertion unless it is explicitly referenced in one of the expressions of that assertion.

NOTE 3—A concurrent assertion statement whose condition is defined by a static expression is equivalent to a process statement that ends in a wait statement that has no sensitivity clause; such a process will execute once through at the beginning of simulation and then wait indefinitely.

NOTE 4—A concurrent statement consisting of the reserved word **assert** followed by a condition, optionally followed by the reserved word **report** and a string expression, is ambiguous. It can be interpreted as a concurrent assertion statement with no severity clause or as a PSL assert directive with a property consisting of a Boolean expression, specifying a condition that shall hold at time zero. The statement is interpreted as a concurrent assertion statement, specifying a condition that shall hold at all times.

11.6 Concurrent signal assignment statements

A concurrent signal assignment statement represents an equivalent process statement that assigns values to signals.

```
concurrent_signal_assignment_statement ::=
      [ label : ] [ postponed ] concurrent_simple_signal_assignment
    | [ label : ] [ postponed ] concurrent_conditional_signal_assignment
    | [ label : ] [ postponed ] concurrent_selected_signal_assignment

concurrent_simple_signal_assignment ::=
    target <= [ guarded ] [ delay_mechanism ] waveform ;

concurrent_conditional_signal_assignment ::=
    target  <=  [ guarded ] [ delay_mechanism ] conditional_waveforms ;

concurrent_selected_signal_assignment ::=
    with expression select [ ? ]
        target <= [ guarded ] [ delay_mechanism ] selected_waveforms ;
```

There are three forms of the concurrent signal assignment statement. For each form, the characteristics that distinguish it are discussed in the following paragraphs.

Each form may include the reserved word **guarded**, which specifies that the signal assignment statement is executed when a signal GUARD changes from FALSE to TRUE, or when that signal has been TRUE and an event occurs on one of the signal assignment statement's inputs. (The signal GUARD shall be either one of the implicitly declared GUARD signals associated with block statements that have guard conditions, or it shall be an explicitly declared signal of type BOOLEAN that is visible at the point of the concurrent signal assignment statement.)

If the target of a concurrent signal assignment is a name that denotes a guarded signal (see 6.4.2.3), or if it is in the form of an aggregate and the expression in each element association of the aggregate is a static signal name denoting a guarded signal, then the target is said to be a *guarded target*. If the target of a concurrent signal assignment is a name that denotes a signal that is not a guarded signal, or if it is in the form of an aggregate and the expression in each element association of the aggregate is a static signal name denoting a signal that is not a guarded signal, then the target is said to be an *unguarded target*. It is an error if the target of a concurrent signal assignment is neither a guarded target nor an unguarded target.

For any concurrent signal assignment statement, there is an equivalent process statement with the same meaning. The process statement equivalent to a concurrent signal assignment statement whose target is a signal name is constructed as follows:

a) If a label appears on the concurrent signal assignment statement, then the same label appears on the process statement.

b) The equivalent process statement is a postponed process if and only if the concurrent signal assignment statement includes the reserved word **postponed**.

c) The statement part of the equivalent process statement consists of a statement transform [described in item e)].

d) If the reserved word **guarded** appears in the concurrent signal assignment statement, then the concurrent signal assignment is called a *guarded assignment*. If the concurrent signal assignment statement is a guarded assignment, and if the target of the concurrent signal assignment is a guarded target, then the statement transform is as follows:

```
if GUARD then
   signal_transform
else
   disconnection_statements
end if;
```

Otherwise, if the concurrent signal assignment statement is a guarded assignment, but if the target of the concurrent signal assignment is *not* a guarded target, then the statement transform is as follows:

```
if GUARD then
   signal_transform
end if;
```

Finally, if the concurrent signal assignment statement is *not* a guarded assignment, and if the target of the concurrent signal assignment is *not* a guarded target, then the statement transform is as follows:

```
signal_transform
```

It is an error if a concurrent signal assignment is not a guarded assignment and the target of the concurrent signal assignment is a guarded target.

A *signal transform* is a sequential signal assignment statement that has no label and that contains a simple, conditional, or selected signal assignment that is the same as the concurrent simple, conditional, or selected signal assignment statement, as appropriate, without the reserved word **guarded**.

e) If the concurrent signal assignment statement is a guarded assignment, or if any expression (other than a time expression) within the concurrent signal assignment statement references a signal, then the process statement contains a final wait statement with an explicit sensitivity clause. The sensitivity clause is constructed by taking the union of the sets constructed by applying the rule of 10.2 to each of the aforementioned expressions. Furthermore, if the concurrent signal assignment statement is a guarded assignment, then the sensitivity clause also contains the simple name GUARD. (The signals identified by these names are called the *inputs* of the signal assignment statement.) Otherwise, the process statement contains a final wait statement that has no explicit sensitivity clause, condition clause, or timeout clause.

Under certain conditions [see item d) in the preceding list] the equivalent process statement may contain a sequence of disconnection statements. A *disconnection statement* is a sequential signal assignment statement that assigns a null transaction to its target. If a sequence of disconnection statements is present in the equivalent process statement, the sequence consists of one sequential signal assignment for each scalar subelement of the target of the concurrent signal assignment statement. For each such sequential signal assignment, the target of the assignment is the corresponding scalar subelement of the target of the concurrent signal assignment, and the waveform of the assignment is a null waveform element whose time expression is given by the applicable disconnection specification (see 7.4).

If the target of a concurrent signal assignment statement is in the form of an aggregate, then the same transformation applies. Such a target shall contain only locally static signal names; moreover, it is an error if any signal is identified by more than one signal name.

It is an error if a null waveform element appears in a waveform of a concurrent signal assignment statement.

Execution of a concurrent signal assignment statement is equivalent to execution of the equivalent process statement.

NOTE 1—A concurrent signal assignment statement whose waveforms and target contain only static expressions is equivalent to a process statement whose final wait statement has no explicit sensitivity clause, so it will execute once through at the beginning of simulation and then suspend permanently.

NOTE 2—A concurrent signal assignment statement whose waveforms are all the reserved word **unaffected** has no drivers for the target, since every waveform in the concurrent signal assignment statement is transformed to the statement

```
null;
```

in the equivalent process statement (see 10.5.2.1).

11.7 Component instantiation statements

11.7.1 General

A component instantiation statement defines a subcomponent of the design entity in which it appears, associates signals or values with the ports of that subcomponent, and associates values with generics of that subcomponent. This subcomponent is one instance of a class of components defined by a corresponding component declaration, design entity, or configuration declaration.

component_instantiation_statement ::=
    *instantiation_*label :
        instantiated_unit
            [generic_map_aspect]
            [port_map_aspect] ;

instantiated_unit ::=
      [**component**] *component_*name
    | **entity** *entity_*name [(*architecture_*identifier)]
    | **configuration** *configuration_*name

The component name, if present, shall be the name of a component declared in a component declaration. The entity name, if present, shall be the name of a previously analyzed entity declaration; if an architecture identifier appears in the instantiated unit, then that identifier shall be the same as the simple name of an architecture body associated with the entity declaration denoted by the corresponding entity name. The architecture identifier defines a simple name that is used during the elaboration of a design hierarchy to select the appropriate architecture body. The configuration name, if present, shall be the name of a

previously analyzed configuration declaration. The generic map aspect, if present, optionally associates a single actual with each local generic (or member thereof) in the corresponding component declaration or entity declaration. Each local generic (or member thereof) shall be associated at most once. Similarly, the port map aspect, if present, optionally associates a single actual with each local port (or member thereof) in the corresponding component declaration or entity declaration. Each local port (or member thereof) shall be associated at most once. The generic map and port map aspects are described in 6.5.7.2 and 6.5.7.3.

If an instantiated unit containing the reserved word **entity** does not contain an explicitly specified architecture identifier, then the architecture identifier is implicitly specified according to the rules given in 7.3.3. The architecture identifier defines a simple name that is used during the elaboration of a design hierarchy to select the appropriate architecture body.

A component instantiation statement and a corresponding configuration specification, if any, taken together, imply that the block hierarchy within the design entity containing the component instantiation is to be extended with a unique copy of the block defined by another design entity. The generic map and port map aspects in the component instantiation statement and in the binding indication of the configuration specification identify the connections that are to be made in order to accomplish the extension.

NOTE 1—A configuration specification can be used to bind a particular instance of a component to a design entity and to associate the local generics and local ports of the component with the formal generics and formal ports of that design entity. A configuration specification can apply to a component instantiation statement only if the name in the instantiated unit of the component instantiation statement denotes a component declaration. See 7.3.

NOTE 2—The component instantiation statement may be used to imply a structural organization for a hardware design. By using component declarations, signals, and component instantiation statements, a given (internal or external) block may be described in terms of subcomponents that are interconnected by signals.

NOTE 3—Component instantiation provides a way of structuring the logical decomposition of a design. The precise structural or behavioral characteristics of a given subcomponent may be described later, provided that the instantiated unit is a component declaration. Component instantiation also provides a mechanism for reusing existing designs in a design library. A configuration specification can bind a given component instance to an existing design entity, even if the generics and ports of the entity declaration do not precisely match those of the component (provided that the instantiated unit is a component declaration); if the generics or ports of the entity declaration do not match those of the component, the configuration specification shall contain a generic map or port map, as appropriate, to map the generics and ports of the entity declaration to those of the component.

11.7.2 Instantiation of a component

A component instantiation statement whose instantiated unit contains a name denoting a component is equivalent to a pair of nested block statements that couple the block hierarchy in the containing design unit to a unique copy of the block hierarchy contained in another design unit (i.e., the subcomponent). The outer block represents the component declaration; the inner block represents the design entity to which the component is bound. Each is defined by a block statement.

The header of the block statement corresponding to the component declaration consists of the generic and port clauses (if present) that appear in the component declaration, followed by the generic map and port map aspects (if present) that appear in the corresponding component instantiation statement. The meaning of any identifier appearing in the header of this block statement is that associated with the corresponding occurrence of the identifier in the generic clause, port clause, generic map aspect, or port map aspect. The statement part of the block statement corresponding to the component declaration consists of a nested block statement corresponding to the design entity.

The header of the block statement corresponding to the design entity consists of the generic and port clauses (if present) that appear in the entity declaration that defines the interface to the design entity, followed by the generic map and port map aspects (if present) that appear in the binding indication that binds the component instance to that design entity. The declarative part of the block statement corresponding to the design entity consists of the declarative items from the entity declarative part, followed by the declarative items from the declarative part of the corresponding architecture body. The statement part of the block statement

corresponding to the design entity consists of the concurrent statements from the entity statement part, followed by the concurrent statements from the statement part of the corresponding architecture body. The meaning of any identifier appearing anywhere in this block statement is that associated with the corresponding occurrence of the identifier in the entity declaration or architecture body.

For example, consider the following component declaration, instantiation, and corresponding configuration specification:

```
component
   COMP port (A,B: inout BIT);
end component;

for C: COMP use
   entity X(Y)
      port map (P1 => A, P2 => B);
   .
   .
   .
C: COMP port map (A => S1, B => S2);
```

Given the following entity declaration and architecture declaration:

```
entity X is
   port (P1, P2: inout BIT);
   constant Delay: TIME := 1 ms;
begin
   CheckTiming (P1, P2, 2*Delay);
end X ;

architecture Y of X is
   signal P3: BIT;
begin
   P3 <= P1 after Delay;
   P2 <= P3 after Delay;
   B: block
      .
      .
      .
   begin
      .
      .
      .
   end block;
end Y;
```

then the following block statements implement the coupling between the block hierarchy in which component instantiation statement C appears and the block hierarchy contained in design entity X(Y):

```
C: block                              --  Component block.
   port (A,B: inout BIT);             --  Local ports.
   port map (A => S1, B => S2);       --  Actual/local binding.
begin
   X: block                           --  Design entity block.
      port (P1, P2 : inout BIT);      --  Formal ports.
```

```
      port map (P1 => A, P2 => B);    --  Local/formal binding.
      constant Delay: TIME := 1 ms;   --  Entity declarative item.
      signal P3: BIT;                --  Architecture declarative item.
   begin
      CheckTiming (P1, P2, 2*Delay); --  Entity statement.
      P3 <= P1 after Delay;           --  Architecture statements.
      P2 <= P3 after Delay;
      B: block                        --  Internal block hierarchy.
         .
         .
         .
      begin
         .
         .
         .
      end block;
   end block X ;
end block C;
```

The block hierarchy extensions implied by component instantiation statements that are bound to design entities are accomplished during the elaboration of a design hierarchy (see Clause 14).

11.7.3 Instantiation of a design entity

A component instantiation statement whose instantiated unit denotes either a design entity or a configuration declaration is equivalent to a pair of nested block statements that couple the block hierarchy in the containing design unit to a unique copy of the block hierarchy contained in another design unit (i.e., the subcomponent). The outer block represents the component instantiation statement; the inner block represents the design entity to which the instance is bound. Each is defined by a block statement.

The header of the block statement corresponding to the component instantiation statement is empty, as is the declarative part of this block statement. The statement part of the block statement corresponding to the component declaration consists of a nested block statement corresponding to the design entity.

The header of the block statement corresponding to the design entity consists of the generic and port clauses (if present) that appear in the entity declaration that defines the interface to the design entity, followed by the generic map and port map aspects (if present) that appear in the component instantiation statement that binds the component instance to a copy of that design entity. The declarative part of the block statement corresponding to the design entity consists of the declarative items from the entity declarative part, followed by the declarative items from the declarative part of the corresponding architecture body. The statement part of the block statement corresponding to the design entity consists of the concurrent statements from the entity statement part, followed by the concurrent statements from the statement part of the corresponding architecture body. The meaning of any identifier appearing anywhere in this block statement is that associated with the corresponding occurrence of the identifier in the entity declaration or architecture body.

For example, consider the following design entity:

```
entity X is
   port (P1, P2: inout BIT);
   constant Delay: DELAY_LENGTH := 1 ms;
   use WORK.TimingChecks.all;
begin
   CheckTiming (P1, P2, 2*Delay);
end entity X;
```

```
architecture Y of X is
   signal P3: BIT;
begin
   P3 <= P1 after Delay;
   P2 <= P3 after Delay;
   B: block
      .
      .
      .
   begin
      .
      .
      .
   end block B;
end architecture Y;
```

This design entity is instantiated by the following component instantiation statement:

```
C: entity WORK.X (Y) port map (P1 => S1, P2 => S2);
```

The following block statements implement the coupling between the block hierarchy in which component instantiation statement C appears and the block hierarchy contained in design entity X(Y):

```
C: block                                -- Instance block.
begin
   X: block                             -- Design entity block.
      port (P1, P2: inout BIT);         -- Entity declaration ports.
      port map (P1 => S1, P2 => S2);    -- Instantiation statement
                                        -- port map.
      constant Delay: DELAY_LENGTH      -- Entity declarative items.
                     := 1 ms;
      use WORK.TimingChecks.all;
      signal P3: BIT;                   -- Architecture declarative item.
   begin
      CheckTiming (P1, P2, 2*Delay);    -- Entity statement.
      P3 <= P1 after Delay;             -- Architecture statements.
      P2 <= P3 after Delay;
      B: block
         .
         .
         .
      begin
         .
         .
         .
      end block B;
   end block X;
end block C;
```

Moreover, consider the following design entity, which is followed by an associated configuration declaration and component instantiation:

```
entity X is
```



```
   port (P1, P2: inout BIT);
   constant Delay: DELAY_LENGTH := 1 ms;
   use WORK.TimingChecks.all;
begin
   CheckTiming (P1, P2, 2*Delay);
end entity X;

architecture Y of X is
   signal P3: BIT;
begin
   P3 <= P1 after Delay;
   P2 <= P3 after Delay;
   B: block
      .
      .
      .
   begin
      .
      .
      .
   end block B;
end architecture Y;
```

The configuration declaration is

```
configuration Alpha of X is
   for Y
      .
      .
      .
   end for;
end configuration Alpha;
```

The component instantiation is

```
C: configuration WORK.Alpha port map (P1 => S1, P2 => S2);
```

The following block statements implement the coupling between the block hierarchy in which component instantiation statement C appears and the block hierarchy contained in design entity X(Y):

```
C: block                                -- Instance block.
begin
   X: block                             -- Design entity block.
      port (P1, P2: inout BIT);         -- Entity declaration ports.
      port map (P1 => S1, P2 => S2);    -- Instantiation statement
                                        -- port map.
      constant Delay: DELAY_LENGTH      -- Entity declarative items.
                    := 1 ms;
      use WORK.TimingChecks.all;
      signal P3: BIT;                   -- Architecture declarative item.
   begin
      CheckTiming (P1, P2, 2*Delay);    -- Entity statement.
      P3 <= P1 after Delay;             -- Architecture statements.
      P2 <= P3 after Delay;
```

```
        B: block
            .
            .
            .
        begin
            .
            .
            .
        end block B;
    end block X;
end block C;
```

The block hierarchy extensions implied by component instantiation statements that are bound to design entities occur during the elaboration of a design hierarchy (see Clause 14).

11.8 Generate statements

A generate statement provides a mechanism for iterative or conditional elaboration of a portion of a description.

generate_statement ::=
    for_generate_statement
  | if_generate_statement
  | case_generate_statement

for_generate_statement ::=
  *generate*_label :
    for *generate*_parameter_specification **generate**
      generate_statement_body
    end generate [*generate*_label] ;

if_generate_statement ::=
  *generate*_label :
    if [*alternative*_label :] condition **generate**
      generate_statement_body
    { **elsif** [*alternative*_label :] condition **generate**
      generate_statement_body }
    [**else** [*alternative*_label :] **generate**
      generate_statement_body]
    end generate [*generate*_label] ;

case_generate_statement ::=
  *generate*_label :
    case expression **generate**
      case_generate_alternative
      { case_generate_alternative }
    end generate [*generate*_label] ;

case_generate_alternative ::=
  when [*alternative*_label :] choices =>
    generate_statement_body

generate_statement_body ::=

[block_declarative_part
begin]
{ concurrent_statement }
[**end** [*alternative*_label] ;]

label ::= identifier

If a label appears at the end of a generate statement, it shall repeat the generate label. The alternative labels, if any, within an if generate statement or a case generate statement shall all be distinct. An alternative label shall not appear at the end of the generate statement body in a for generate statement. If a label appears at the end of a generate statement body in an if generate statement, then the immediately enclosing **if**, **elsif**, or **else** part of the if generate statement shall include an alternative label, and the label at the end of the generate statement body shall repeat the alternative label. Similarly, if a label appears at the end of a generate statement body in a case generate statement, then the immediately enclosing case generate alternative of the case generate statement shall include an alternative label, and the label at the end of the generate statement body shall repeat the alternative label.

For a for generate statement, the generate parameter specification is the declaration of the *generate parameter* with the given identifier. The generate parameter is a constant object whose type is the base type of the discrete range of the generate parameter specification.

The discrete range in the generate parameter specification of a for generate statement shall be a static discrete range; similarly, each condition in an if generate statement shall be a static expression.

For a case generate statement, the expression shall be globally static, and shall be of a discrete type, or of a one-dimensional array type whose element base type is a character type. This type shall be determined by applying the rules of 12.5 to the expression considered as a complete context, using the fact that the expression shall be of a discrete type or a one-dimensional character array type. Each choice in a case generate alternative shall be of the same type as the expression; the list of choices specifies for which values of the expression the alternative is chosen.

If the expression is the name of an object whose subtype is globally static, whether a scalar type or an array type, then each value of the subtype shall be represented once and only once in the set of choices of the case generate statement, and no other value is allowed; this rule is likewise applied if the expression is a qualified expression or type conversion whose type mark denotes a globally static subtype, or if the expression is a call to a function whose return type mark denotes a globally static subtype, or if the expression is an expression described in this paragraph and enclosed in parentheses.

If the expression is of a one-dimensional character array type and is not described by the preceding paragraph, then the values of all of the choices, except the **others** choice, if present, shall be of the same length. Moreover, each value of the (base) type of the expression shall be represented once and only once in the set of choices, and no other value is allowed. It is an error if the value of the expression is not of the same length as the values of the choices. If there is only one choice and that choice is **others**, then the value of the expression may be of any length.

For other forms of expression, each value of the (base) type of the expression shall be represented once and only once in the set of choices, and no other value is allowed.

The simple expression and discrete ranges given as choices in a case generate statement shall be globally static. A choice defined by a discrete range stands for all values in the corresponding range. The choice **others** is only allowed for the last alternative and as its only choice; it stands for all values (possibly none) not given in the choices of previous alternatives. An element simple name (see 9.3.3.1) is not allowed as a choice of a case generate alternative.

The elaboration of a generate statement is described in 14.5.3.

Example:

```
Gen: block
begin
   L1: CELL port map (Top, Bottom, A(0), B(0));
   L2: for I in 1 to 3 generate
      L3: for J in 1 to 3 generate
         L4: if I+J>4 generate
            L5: CELL port map (A(I-1),B(J-1),A(I),B(J));
         end generate;
      end generate;
   end generate;

   L6: for I in 1 to 3 generate
      L7: for J in 1 to 3 generate
         L8: if I+J<4 generate
            L9: CELL port map (A(I+1),B(J+1),A(I),B(J));
         end generate;
      end generate;
   end generate;
end block Gen;

Gen2: block
begin
   L1: case verify_mode generate
      when V_rtl: all_rtl | cpu_rtl =>
         CPU1: entity work.cpu(rtl) port map ( ... );
      when V_bfm: others =>
            signal bfm_sig : BIT;
         begin
            CPU1: entity work.cpu(bfm) port map ( ... );
         end V_bfm;
   end generate L1;

   L2: if A1: max_latency < 10 generate
         signal s1 : BIT;
      begin
         multiplier1: parallel_multiplier port map ( ... );
      end A1;
   else A2: generate
         signal s1 : STD_LOGIC;
      begin
         multiplier1: sequential_multiplier port map ( ... );
      end A2;
   end generate L2;
end block Gen2;
```


12. Scope and visibility

12.1 Declarative region

With two exceptions, a declarative region is a portion of the text of the description. A single declarative region is formed by the text of each of the following:

a) An entity declaration, together with a corresponding architecture body
b) A configuration declaration
c) A subprogram declaration, together with the corresponding subprogram body
d) A package declaration together with the corresponding body (if any)
e) A record type declaration
f) A component declaration
g) A block statement
h) A process statement
i) A loop statement
j) A block configuration
k) A component configuration
l) A generate statement
m) A protected type declaration, together with the corresponding body

In each of these cases, the declarative region is said to be *associated* with the corresponding declaration or statement. A declaration is said to occur *immediately within* a declarative region if this region is the innermost region that encloses the declaration, not counting the declarative region (if any) associated with the declaration itself.

Certain declarative regions include disjoint parts. Each declarative region is nevertheless considered as a (logically) continuous portion of the description text. Hence, if any rule defines a portion of text as the text that *extends* from some specific point of a declarative region to the end of this region, then this portion is the corresponding subset of the declarative region (thus, it does not include intermediate declarative items between the interface declaration and a corresponding body declaration).

In addition to the preceding declarative regions, there is a *root declarative region*, not associated with a portion of the text of the description, but encompassing any given primary unit. At the beginning of the analysis of a given primary unit, there are no declarations whose scopes (see 12.2) are within the root declarative region. Moreover, the root declarative region associated with any given secondary unit is the root declarative region of the corresponding primary unit.

There is also a *library declarative region* associated with each design library (see 13.2). Each library declarative region has within its scope declarations corresponding to each primary unit contained within the associated design library.

NOTE—An architecture body, though a declaration, does not occur immediately within any declarative region.

12.2 Scope of declarations

For each form of declaration, the language rules define a certain portion of the description text called the *scope of the declaration*. The scope of a declaration is also called the scope of any named entity declared by the declaration. Furthermore, if the declaration associates some notation (either an identifier, a character literal, or an operator symbol) with the named entity, this portion of the text is also called the scope of this notation. Within the scope of a named entity, and only there, there are places where it is legal to use the

associated notation in order to refer to the named entity. These places are defined by the rules of visibility and overloading.

The scope of a declaration, except for an architecture body, extends from the beginning of the declaration to the end of the immediately closing declarative region; the scope of an architecture body extends from the beginning to the end of the architecture body. In either case, this part of the scope of a declaration is called the *immediate scope*. Furthermore, for any of the declarations in the following list, the scope of the declaration extends beyond the immediate scope:

a) A declaration that occurs immediately within a package declaration

b) An element declaration in a record type declaration

c) A formal parameter declaration in a subprogram declaration

d) A local generic declaration in a component declaration

e) A local port declaration in a component declaration

f) A formal generic declaration in an entity declaration, an uninstantiated package declaration, or an uninstantiated subprogram declaration

g) A formal port declaration in an entity declaration

h) A declaration that occurs immediately within a protected type declaration

i) An architecture body

In the absence of a separate subprogram declaration, the subprogram specification given in the subprogram body acts as the declaration, and rule c) applies also in such a case. In each of these cases except i), the given declaration occurs immediately within some enclosing declaration, and the scope of the given declaration extends to the end of the scope of the enclosing declaration.

In addition to the preceding rules, if the the scope of any declaration includes the end of the declarative part of a given block (whether it be an external block defined by a design entity or an internal block defined by a block statement) then the scope of the declaration extends into a configuration declaration that configures the given block.

If a component configuration appears as a configuration item immediately within a block configuration that configures a given block, and if the scope of a given declaration includes the end of the declarative part of that block, then the scope of the given declaration extends from the beginning to the end of the declarative region associated with the given component configuration. A similar rule applies to a block configuration that appears as a configuration item immediately within another block configuration, provided that the contained block configuration configures an internal block. Furthermore, the scope of a use clause is similarly extended. Finally, the scope of a library unit contained within a design library is extended along with the scope of the logical library name corresponding to that design library.

If the scope of any declaration includes the end of the declarative region of the design entity at the root of the design hierarchy, then the scope extends into a PSL verification unit that is bound to that design entity. Similarly, if the scope of any declaration includes the end of the declarative region of a design entity bound to a component instance, then the scope extends into a PSL verification unit that is bound to that component instance.

NOTE 1—These scope rules apply to all forms of declaration. In particular, they apply also to implicit declarations and to named primary units.

NOTE 2—The scope of an entity declaration includes an associated architecture body, if any. Thus, the entity name may be used within the architecture body as the prefix of an expanded name denoting a declaration that occurs immediately within the entity declaration or the architecture body. The scope of an architecture body does not include the corresponding entity declaration. Thus, the entity cannot use an expanded name to refer to the architecture body nor to any declaration within the architecture body.

12.3 Visibility

The meaning of the occurrence of an identifier at a given place in the text is defined by the visibility rules and also, in the case of overloaded declarations, by the overloading rules. The identifiers considered in this subclause include any identifier other than a reserved word or an attribute designator that denotes a predefined attribute. The places considered in this subclause are those where a lexical element (such as an identifier) occurs. The overloaded declarations considered in this subclause are those for subprograms and enumeration literals.

For each identifier and at each place in the text, the visibility rules determine a set of declarations (with this identifier) that define the possible meanings of an occurrence of the identifier. A declaration is said to be *visible* at a given place in the text when, according to the visibility rules, the declaration defines a possible meaning of this occurrence. The following two cases arise in determining the meaning of such a declaration:

— The visibility rules determine *at most one* possible meaning. In such a case, the visibility rules are sufficient to determine the declaration defining the meaning of the occurrence of the identifier, or in the absence of such a declaration, to determine that the occurrence is not legal at the given point.

— The visibility rules determine *more than one* possible meaning. In such a case, the occurrence of the identifier is legal at this point if and only if *exactly one* visible declaration is acceptable for the overloading rules in the given context or all visible declarations denote the same named entity.

A declaration is visible only within a certain part of its scope; this part starts at the end of the declaration except in the declaration of a design unit other than a PSL verification unit, a package declaration, or a protected type declaration, in which case it starts immediately after the reserved word **is** occurring after the identifier of the design unit, a package declaration, or protected type declaration. This rule applies to both explicit and implicit declarations.

Visibility is either by selection or direct. A declaration is visible *by selection* at places that are defined as follows:

a) For a primary unit contained in a library: at the place of the suffix in a selected name whose prefix denotes the library.

b) For an entity name in a configuration declaration whose entity name is a simple name: at the place of the simple name, and the context is that of the library WORK.

c) For an architecture body associated with a given entity declaration: at the place of the block specification in a block configuration for an external block whose interface is defined by that entity declaration.

d) For an architecture body associated with a given entity declaration: at the place of an architecture identifier (between the parentheses) in the first form of an entity aspect in a binding indication.

e) For an architecture body associated with a given entity declaration: at the place of an architecture identifier (between the parentheses) in the second form of an instantiated unit in a component instantiation statement.

f) For a declaration given in a package declaration, other than in a package declaration that defines an uninstantiated package: at the place of the suffix in a selected name whose prefix denotes the package.

g) For an element declaration of a given record type declaration: at the place of the suffix in a selected name whose prefix is appropriate for the type; also at the place of a choice (before the compound delimiter =>) in a named element association of an aggregate of the type.

h) For an element declaration of a given record type declaration: at the place of the record element simple name in a record element constraint of a record constraint that applies to a type or subtype that is the given record type or an access type whose designated type is the given record type; also at the place of a record element simple name in a record element resolution of a record resolution corresponding to the given record type or a subtype of the given record type.

i) For a user-defined attribute: at the place of the attribute designator (after the delimiter ') in an attribute name whose prefix denotes a named entity with which that attribute has been associated.

j) For a formal parameter declaration of a given subprogram declaration: at the place of the formal part (before the compound delimiter =>) of a named parameter association element of a corresponding subprogram call.

k) For a local generic declaration of a given component declaration: at the place of the formal part (before the compound delimiter =>) of a named generic association element of a corresponding component instantiation statement; similarly, at the place of the actual part (after the compound delimiter =>, if any) of a generic association element of a corresponding binding indication.

l) For a local port declaration of a given component declaration: at the place of the formal part (before the compound delimiter =>) of a named port association element of a corresponding component instantiation statement; similarly, at the place of the actual part (after the compound delimiter =>, if any) of a port association element of a corresponding binding indication.

m) For a formal generic declaration of a given entity declaration: at the place of the formal part (before the compound delimiter =>) of a named generic association element of a corresponding binding indication; similarly, at the place of the formal part (before the compound delimiter =>) of a generic association element of a corresponding component instantiation statement when the instantiated unit is a design entity or a configuration declaration.

n) For a formal port declaration of a given entity declaration: at the place of the formal part (before the compound delimiter =>) of a named port association element of a corresponding binding indication; similarly, at the place of the formal part (before the compound delimiter =>) of a port association element of a corresponding component instantiation statement when the instantiated unit is a design entity or a configuration declaration.

o) For a formal generic declaration or a formal port declaration of a given block statement: at the place of the formal part (before the compound delimiter =>) of a named association element of a corresponding generic or port map aspect.

p) For a formal generic declaration of a given package declaration: at the place of the formal part (before the compound delimiter =>) of a named association element of a corresponding generic map aspect.

q) For a formal generic declaration of a given subprogram declaration: at the place of the formal part (before the compound delimiter =>) of a named association element of a corresponding generic map aspect.

r) For a formal generic type of a given uninstantiated subprogram declaration: at the place of a signature in a subprogram instantiation declaration in which the uninstantiated subprogram name denotes the given uninstantiated subprogram declaration.

s) For a subprogram declared immediately within a given protected type declaration: at the place of the suffix in a selected name whose prefix denotes an object of the protected type.

t) For an alternative label of an if generate statement or a case generate statement: at the place of the generate specification in a block specification that refers to the generate statement label of the generate statement.

Finally, within the declarative region associated with a construct other than a record type declaration or a protected type, any declaration that occurs immediately within the region and that also occurs textually within the construct is visible by selection at the place of the suffix of an expanded name whose prefix denotes the construct. Similarly, within an architecture body, any declaration that occurs immediately within the architecture body or the corresponding entity declaration is visible by selection at the place of the suffix of an expanded name whose prefix denotes the entity declaration.

Where it is not visible by selection, a visible declaration is said to be *directly visible*. A declaration is said to be directly visible within a certain part of its immediate scope; this part extends to the end of the immediate scope of the declaration but excludes places where the declaration is hidden as explained in the following

paragraphs. In addition, a declaration occurring immediately within the visible part of a package, other than an uninstantiated package, can be made directly visible by means of a use clause according to the rules described in 12.4.

A declaration is said to be *hidden* within (part of) an inner declarative region if the inner region contains a homograph of this declaration; the outer declaration is then hidden within the immediate scope of the inner homograph. Each of two declarations is said to be a *homograph* of the other if and only if both declarations have the same designator, and they denote different named entities, and either overloading is allowed for at most one of the two, or overloading is allowed for both declarations and they have the same parameter and result type profile (see 4.5.1).

At a place in which a given declaration is visible by selection, every declaration with the same designator as the given declaration and that would otherwise be directly visible is hidden.

Within the specification of a subprogram, every declaration with the same designator as the subprogram is hidden. Where hidden in this manner, a declaration is visible neither by selection nor directly.

Two declarations that occur immediately within the same declarative region, other than the declarative region of a block implied by a component instantiation or the declarative region of a generic-mapped package or subprogram equivalent to a package instance or a subprogram instance, shall not be homographs, unless exactly one of them is the implicit declaration of a predefined operation or is an implicit alias of such an implicit declaration. In such cases, a predefined operation or alias thereof is always hidden by the other homograph. Where hidden in this manner, an implicit declaration is hidden within the entire scope of the other declaration (regardless of which declaration occurs first); the implicit declaration is visible neither by selection nor directly. For a declarative region of a block implied by a component instantiation or the declarative region of a generic-mapped package or subprogram equivalent to a package instance or a subprogram instance, the rules of this paragraph are applied to the corresponding entity declaration, component declaration, uninstantiated package declaration, or uninstantiated subprogram declaration, as appropriate.

A declaration is hidden within a PSL declaration, a PSL directive, or a PSL verification unit if the simple name of the declaration is a PSL keyword.

Whenever a declaration with a certain identifier is visible from a given point, the identifier and the named entity (if any) are also said to be visible from that point. Direct visibility and visibility by selection are likewise defined for character literals and operator symbols. An operator is directly visible if and only if the corresponding operator declaration is directly visible.

In addition to the aforementioned rules, any declaration that is visible by selection at the end of the declarative part of a given (external or internal) block is visible by selection in a configuration declaration that configures the given block.

In addition, any declaration that is directly visible at the end of the declarative part of a given block is directly visible in a block configuration that configures the given block. This rule holds unless a use clause that makes a homograph of the declaration potentially visible (see 12.4) appears in the corresponding configuration declaration, and if the scope of that use clause encompasses all or part of those configuration items. If such a use clause appears, then the declaration will be directly visible within the corresponding configuration items, except at those places that fall within the scope of the additional use clause. At such places, neither name will be directly visible.

If a component configuration appears as a configuration item immediately within a block configuration that configures a given block, and if a given declaration is visible by selection at the end of the declarative part of that block, then the given declaration is visible by selection from the beginning to the end of the declarative region associated with the given component configuration. A similar rule applies to a block configuration

that appears as a configuration item immediately within another block configuration, provided that the contained block configuration configures an internal block.

If a component configuration appears as a configuration item immediately within a block configuration that configures a given block, and if a given declaration is directly visible at the end of the declarative part of that block, then the given declaration is visible by selection from the beginning to the end of the declarative region associated with the given component configuration. A similar rule applies to a block configuration that appears as a configuration item immediately within another block configuration, provided that the contained block configuration configures an internal block. Furthermore, the visibility of declarations made directly visible by a use clause within a block is similarly extended. Finally, the visibility of a logical library name corresponding to a design library directly visible at the end of a block is similarly extended. The rules of this paragraph hold unless a use clause that makes a homograph of the declaration potentially visible appears in the corresponding block configuration, and if the scope of that use clause encompasses all or part of those configuration items. If such a use clause appears, then the declaration will be directly visible within the corresponding configuration items, except at those places that fall within the scope of the additional use clause. At such places, neither name will be directly visible.

NOTE 1—The same identifier, character literal, or operator symbol may occur in different declarations and may thus be associated with different named entities, even if the scopes of these declarations overlap. Overlap of the scopes of declarations with the same identifier, character literal, or operator symbol can result from overloading of subprograms and of enumeration literals. Such overlaps can also occur for named entities declared in the visible parts of packages and for formal generics and ports, record elements, and formal parameters, where there is overlap of the scopes of the enclosing package declarations, entity declarations, record type declarations, or subprogram declarations. Finally, overlapping scopes can result from nesting.

NOTE 2—The rules defining immediate scope, hiding, and visibility imply that a reference to an identifier, character literal, or operator symbol within its own declaration is illegal (except for design units). The identifier, character literal, or operator symbol hides outer homographs within its immediate scope—that is, from the start of the declaration. On the other hand, the identifier, character literal, or operator symbol is visible only after the end of the declaration (again, except for design units). For this reason, all but the last of the following declarations are illegal:

```
constant K: INTEGER := K*K;              --  Illegal
constant T: T;                           --  Illegal
procedure P (X: P);                      --  Illegal
function Q (X: REAL := Q) return Q;      --  Illegal
procedure R (R: REAL);                   --  Legal (although perhaps confusing)
```

NOTE 3—A declaration in an uninstantiated package cannot be made visible by selection by referencing it with a selected name. However, a declaration in an instance of the package can be referenced with a selected name.

NOTE 4—There are circumstances where it is legal for two subprograms declared in the same declarative region to be homographs. An example is the declaration of the following two subprograms in an uninstantiated package with formal generic types T1 and T2:

procedure P (X: T1);
procedure P (X: T2);

Since T1 and T2 are distinct types, the subprograms are not homographs within the uninstantiated package. If an instance of the package associates the same actual type with both T1 and T2, then the subprograms are legal homographs within the instance. However, any call to either of the subprograms in the instance will be ambiguous.

NOTE 5—The visibility of declarations within a PSL verification unit is defined in IEEE Std 1850-2005.

Example:

```
L1: block
   signal A,B: Bit;
begin
   L2: block
      signal B: Bit;                     --  An inner homograph of B.
   begin
```



```
      A <= B after 5 ns;                -- Means L1.A <= L2.B
      B <= L1.B after 10 ns;            -- Means L2.B <= L1.B
   end block ;
   B <= A after 15 ns;                  -- Means L1.B <= L1.A
end block;
```

12.4 Use clauses

A use clause achieves direct visibility of declarations that are visible by selection.

use_clause ::=
    use selected_name { , selected_name } ;

Each selected name in a use clause identifies one or more declarations that will potentially become directly visible. If the suffix of the selected name is a simple name other than a type mark, or is a character literal or operator symbol, then the selected name identifies only the declaration(s) of that simple name, character literal, or operator symbol contained within the package or library denoted by the prefix of the selected name.

If the suffix of the selected name is a type mark, then the declaration of the type or subtype denoted by the type mark is identified. Moreover, the following declarations, if any, that occur immediately within the package denoted by the prefix of the selected name, are also identified:

- If the type mark denotes an enumeration type or a subtype of an enumeration type, the enumeration literals of the base type
- If the type mark denotes a subtype of a physical type, the units of the base type
- The implicit declarations of predefined operations for the type that are not hidden by homographs explicitly declared immediately within the package denoted by the prefix of the selected name
- The declarations of homographs, explicitly declared immediately within the package denoted by the prefix of the selected name, that hide implicit declarations of predefined operations for the type

If the suffix is the reserved word **all**, then the selected name identifies all declarations that are contained within the package or library denoted by the prefix of the selected name.

It is an error if the prefix of a selected name in a use clause denotes an uninstantiated package.

For each use clause, except a use clause that appears within a context declaration, there is a certain region of text called the *scope* of the use clause. This region starts immediately after the use clause. If a use clause is a declarative item of some declarative region, the scope of the clause extends to the end of the given declarative region. If a use clause occurs within the context clause of a design unit, the scope of the use clause extends to the end of the root declarative region associated with the given design unit. The scope of a use clause may additionally extend into a configuration declaration (see 12.2).

In order to determine which declarations are made directly visible at a given place by use clauses, consider the set of declarations identified by all use clauses whose scopes enclose this place. Any declaration in this set is a potentially visible declaration. A potentially visible declaration is actually made directly visible except in the following three cases:

a) A potentially visible declaration is not made directly visible if the place considered is within the immediate scope of a homograph of the declaration.

b) If two potentially visible declarations are homographs and one is explicitly declared and the other is implicitly declared, then the implicit declaration is not made directly visible.

c) Potentially visible declarations that have the same designator and that are not covered by case b) are not made directly visible unless each of them is either an enumeration literal specification or the declaration of a subprogram.

NOTE 1—These rules guarantee that a declaration that is made directly visible by a use clause cannot hide an otherwise directly visible declaration. Moreover, an explicitly declared operation has priority over an implicitly declared homograph of that operation if both are made potentially visible by use clauses.

NOTE 2—If a named entity X declared in package P is made potentially visible within a package Q (e.g., by the inclusion of the clause "**use** P.X;" in the context clause of package Q), and the context clause for design unit R includes the clause "**use** Q.**all**;", this does not imply that X will be potentially visible in R. Only those named entities that are actually declared in package Q will be potentially visible in design unit R (in the absence of any other use clauses).

NOTE 3—A declaration in an uninstantiated package cannot be made potentially or directly visible by a use clause. However, a declaration in an instance of the package can be made potentially or directly visible by a use clause.

12.5 The context of overload resolution

Overloading is defined for names, subprograms, and enumeration literals.

For overloaded entities, overload resolution determines the actual meaning that an occurrence of an identifier or a character literal has whenever the visibility rules have determined that more than one meaning is acceptable at the place of this occurrence; overload resolution likewise determines the actual meaning of an occurrence of an operator or basic operation (see 5.1).

At such a place, all visible declarations are considered. The occurrence is only legal if there is exactly one interpretation of each constituent of the innermost *complete context*. Each of the following constructs is a complete context:

— A declaration
— A specification
— A statement
— A discrete range used in a constrained array definition, a generate parameter specification, or a loop parameter specification
— The expression of a type conversion
— The expression of a case statement or a case generate statement
— The expression following a for generate statement label in an external name

When considering possible interpretations of a complete context, the only rules considered are the syntax rules, the scope and visibility rules, and the rules of the form as follows:

a) Any rule that requires a name or expression to have a certain type or to have the same type as another name or expression.
b) Any rule that requires the type of a name or expression to be a type of a certain class; similarly, any rule that requires a certain type to be a discrete, integer, floating-point, physical, universal, or character type.
c) Any rule that requires a prefix to be appropriate for a certain type.
d) The rules that require the type of an aggregate or string literal to be determinable solely from the enclosing complete context. Similarly, the rules that require that the meaning of the prefix of an attribute must be determinable independently of the attribute designator and independently of the fact that it is the prefix of an attribute.
e) The rules given for the resolution of overloaded subprogram calls; for the implicit conversions of universal expressions; for the interpretation of discrete ranges with bounds having a universal type; for the interpretation of an expanded name whose prefix denotes a subprogram; and for a

subprogram named in a subprogram instantiation declaration to denote an uninstantiated subprogram.

f) The rules given for the requirements on the return type, the number of formal parameters, and the types of the formal parameters of the subprogram denoted by the resolution function name (see 4.6).

NOTE 1—If there is only one possible interpretation of an occurrence of an identifier, character literal, operator symbol, or string, that occurrence denotes the corresponding named entity. However, this condition does not mean that the occurrence is necessarily legal since other requirements exist that are not considered for overload resolution: for example, the fact that the expression is static, the parameter modes, conformance rules, the use of named association in an indexed name, the use of **open** in an indexed name, the use of a slice as an actual to a function call, and so forth.

NOTE 2—A loop parameter specification is a declaration, and hence a complete context.

NOTE 3—Rules that require certain constructs to have the same parameter and result type profile fall under the preceding category a). This includes the rule that the actual associated with a formal generic subprogram have a conforming profile with the formal. The same holds for rules that require lexical conformance of two constructs, since lexical conformance requires that corresponding names be given the same meaning by the visibility and overloading rules.

13. Design units and their analysis

13.1 Design units

Certain constructs are independently analyzed and inserted into a design library; these constructs are called *design units*. One or more design units in sequence comprise a *design file*.

design_file ::= design_unit { design_unit }

design_unit ::= context_clause library_unit

library_unit ::=
    primary_unit
  | secondary_unit

primary_unit ::=
    entity_declaration
  | configuration_declaration
  | package_declaration
  | package_instantiation_declaration
  | context_declaration
  | *PSL*_Verification_Unit

secondary_unit ::=
    architecture_body
  | package_body

Design units in a design file are analyzed in the textual order of their appearance in the design file. Analysis of a design unit defines the corresponding library unit in a design library. A *library unit* is either a primary unit or a secondary unit. A secondary unit is a separately analyzed body of a primary unit resulting from a previous analysis.

It is an error if the context clause preceding a library unit that is a context declaration is not empty.

The name of a primary unit is given by the first identifier after the initial reserved word of that unit. Of the secondary units, only architecture bodies are named; the name of an architecture body is given by the identifier following the reserved word **architecture**. Each primary unit in a given library shall have a simple name that is unique within the given library, and each architecture body associated with a given entity declaration shall have a simple name that is unique within the set of names of the architecture bodies associated with that entity declaration.

Entity declarations, architecture bodies, and configuration declarations are discussed in Clause 3. Package declarations, package bodies, and package instantiations are discussed in Clause 4. Context declarations are discussed in 13.3. PSL verification units are described in IEEE Std 1850-2005.

13.2 Design libraries

A *design library* is an implementation-dependent storage facility for previously analyzed design units. A given implementation is required to support any number of design libraries.

library_clause ::= **library** logical_name_list ;

logical_name_list ::= logical_name { , logical_name }

logical_name ::= identifier

A library clause defines logical names for design libraries in the host environment. A library clause appears as part of a context clause, either at the beginning of a design unit or within a context declaration. For the former case, the declaration of each logical name defined by the library clause occurs immediately within the root declarative region associated with the design unit. For a library clause that appears within a context declarative region, the logical names are not declared; rather, there is an equivalent library clause that declares the logical names (see 13.4).

If two or more logical names having the same identifier (see 15.4) appear in library clauses in the same context clause, the second and subsequent occurrences of the logical name have no effect. The same is true of logical names appearing both in the context clause of a primary unit and in the context clause of a corresponding secondary unit.

Each logical name defined by the library clause denotes a design library in the host environment.

For a given library logical name, the actual name of the corresponding design library in the host environment may or may not be the same. A given implementation shall provide some mechanism to associate a library logical name with a host-dependent library. Such a mechanism is not defined by the language.

There are two classes of design libraries: *working libraries* and *resource libraries*. A working library is the library into which the library unit resulting from the analysis of a design unit is placed. A resource library is a library containing library units that are referenced within the design unit being analyzed. Only one library is the working library during the analysis of any given design unit; in contrast, any number of libraries (including the working library itself) may be resource libraries during such an analysis.

Every design unit except a context declaration and package STANDARD is assumed to contain the following implicit context items as part of its context clause:

```
library STD, WORK; use STD.STANDARD.all;
```

Library logical name STD denotes the design library in which packages STANDARD, TEXTIO, and ENV reside (see Clause 16). (The use clause makes all declarations within package STANDARD directly visible within the corresponding design unit; see 12.4.) Library logical name WORK denotes the current working library during a given analysis. Library logical name IEEE denotes the design library in which the mathematical, multivalue logic and synthesis packages, and the synthesis context declarations reside (see Clause 16).

The library denoted by the library logical name STD contains no library units other than packages STANDARD, TEXTIO, and ENV.

A secondary unit corresponding to a given primary unit shall be placed into the design library in which the primary unit resides.

NOTE—The design of the language assumes that the contents of resource libraries named in all library clauses in the context clause of a design unit will remain unchanged during the analysis of that unit (with the possible exception of the updating of the library unit corresponding to the analyzed design unit within the working library, if that library is also a resource library).

13.3 Context declarations

A context declaration defines context items that may be referenced by design units.

context_declaration ::=
    context identifier **is**
        context_clause
    end [**context**] [*context*_simple_name] ;

If a simple name appears at the end of a context declaration, it shall repeat the identifier of the context declaration.

It is an error if a library clause in a context declaration defines the library logical name WORK, or if a selected name in a use clause or a context reference in a context declaration has the library logical name WORK as a prefix.

Example:

```
context project_context is
   library project_lib;
   use project_lib.project_defs.all;
   library IP_lib;
   context IP_lib.IP_context;
end context project_context;
```

13.4 Context clauses

A context clause defines the initial name environment in which a design unit is analyzed.

context_clause ::= { context_item }

context_item ::=
      library_clause
    | use_clause
    | context_reference

context_reference ::=
    context selected_name { , selected_name } ;

A library clause defines library logical names that may be referenced in the design unit; library clauses are described in 13.2. A use clause makes certain declarations directly visible within the design unit; use clauses are described in 12.4.

It is an error if a selected name in a context reference does not denote a context declaration.

A given context clause is equivalent to an expanded context clause containing only library clauses and use clauses. The expanded context clause is formed from the given context clause by replacing each context reference with the expanded context clause of the context clause in the context declaration denoted by the selected name of the context reference.

For a context clause that precedes a library unit, rules concerning scope and visibility are interpreted to apply to the expanded context clause at the place of the context clause.

It is an error if, during analysis of a design unit, there is a library clause in the expanded context clause of the design unit that occurs as part of a replacement of a context reference, and a logical name in that library clause denotes a different design library from the design library denoted by the logical name during analysis of the context declaration from which the library clause was expanded.

NOTE 1—The rules given for use clauses are such that the same effect is obtained whether the name of a library unit is mentioned once or more than once by the applicable use clauses, or even within a given use clause.

NOTE 2—For a context clause that appears within a context declaration, the library clauses and use clauses have no scope; hence, rules concerning scope and visibility do not apply.

13.5 Order of analysis

The rules defining the order in which design units can be analyzed are direct consequences of the visibility rules. In particular

a) A primary unit whose name is referenced within a given design unit shall be analyzed prior to the analysis of the given design unit.
b) A primary unit shall be analyzed prior to the analysis of any corresponding secondary unit.

In each case, the second unit *depends* on the first unit.

The order in which design units are analyzed shall be consistent with the partial ordering defined by the preceding rules.

If any error is detected while attempting to analyze a design unit, then the attempted analysis is rejected and has no effect whatsoever on the current working library.

A given library unit is potentially affected by a change in any library unit whose name is referenced within the given library unit. A secondary unit is potentially affected by a change in its corresponding primary unit. If a library unit is changed (e.g., by reanalysis of the corresponding design unit), then all library units that are potentially affected by such a change become obsolete and shall be reanalyzed before they can be used again.

14. Elaboration and execution

14.1 General

The process by which a declaration achieves its effect is called the *elaboration* of the declaration. After its elaboration, a declaration is said to be elaborated. Prior to the completion of its elaboration (including before the elaboration), the declaration is not yet elaborated.

Elaboration is also defined for design hierarchies, declarative parts, statement parts (containing concurrent statements), and concurrent statements. Elaboration of such constructs is necessary in order ultimately to elaborate declarative items that are declared within those constructs.

In order to execute a model, the design hierarchy defining the model shall first be elaborated. Initialization of nets (see 14.7.3.4) in the model then occurs. Finally, simulation of the model proceeds. Simulation consists of the repetitive execution of the *simulation cycle*, during which processes are executed and nets updated.

14.2 Elaboration of a design hierarchy

The elaboration of a design hierarchy creates a collection of processes interconnected by nets; this collection of processes and nets can then be executed to simulate the behavior of the design.

At the beginning of the elaboration of a design hierarchy, every registered and enabled `vhpiCbStartOfElaboration` callback is executed. Once the elaboration of a given design hierarchy is complete, every registered and enabled `vhpiCbEndOfElaboration` callback is executed.

A design hierarchy is defined either by a design entity or by a configuration.

An implementation may allow PSL verification units, in addition to any whose binding is specified as part of the design hierarchy, to be bound to design entities within the design hierarchy. The manner in which such PSL verification units are identified and the manner in which binding is specified for such PSL verification units that are not explicitly bound are not defined by this standard.

Elaboration of a design hierarchy defined by a design entity consists of the elaboration of the block statement equivalent to the external block defined by the design entity. The architecture of this design entity is assumed to contain an implicit configuration specification (see 7.3) for each component instance that is unbound in this architecture; each configuration specification has an entity aspect denoting an anonymous configuration declaration identifying the visible entity declaration (see 7.3.3) and supplying an implicit block configuration (see 3.4.2) that binds and configures a design entity identified according to the rules of 7.3.3. The equivalent block statement is defined in 11.7.3. Elaboration of a block statement is defined in 14.5.2.

Elaboration of a configuration consists of the elaboration of the block statement equivalent to the external block defined by the design entity configured by the configuration. The configuration contains an implicit component configuration (see 3.4.3) for each unbound component instance contained within the external block and an implicit block configuration (see 3.4.2) for each internal block contained within the external block.

An implementation may allow, but is not required to allow, a design entity at the root of a design hierarchy to have generics and ports. If an implementation allows these *top-level* interface objects, it may restrict their allowed forms (that is, whether they are allowed to be interface types, subprograms, packages, or objects), and, in the case of interface objects, their allowed types and modes in an implementation-defined manner.

Similarly, the means by which top-level interface objects are associated with the external environment of the hierarchy are also defined by an implementation supporting top-level interface objects.

Elaboration of a block statement involves first elaborating each not-yet-elaborated package primary unit or package instantiation primary unit containing declarations referenced by the block. Similarly, elaboration of a given package primary unit or package instantiation primary unit involves first elaborating each not-yet-elaborated package primary unit or package instantiation primary unit containing declarations referenced by the given package or package instantiation. Elaboration of a package primary unit consists additionally of the following:

a) Elaboration of the package declaration, eventually followed by

b) Elaboration of the corresponding package body, if the package has a corresponding package body.

Elaboration of a package instantiation primary unit consists of elaboration of the equivalent generic-mapped package declaration, eventually followed by elaboration of the corresponding equivalent generic-mapped package body, if such a package body is defined (see 4.9).

Step b), the elaboration of a package body, may be deferred until the declarations of other packages have been elaborated, if necessary, because of the dependencies created between packages by their interpackage references. Similarly, elaboration of an equivalent generic-mapped package body may be deferred if necessary.

Elaboration of a package is defined in 14.4.2.9.

For a block statement implied by a design entity, whether the design entity at the root of the design hierarchy or a design entity bound to a component instance, to which one or more PSL verification units are bound, after elaboration of the implied block statement, each PSL verification unit bound to the design entity is elaborated. Elaboration of a PSL verification unit involves first elaborating each not-yet-elaborated package primary unit or package instantiation primary unit containing declarations referenced by the PSL verification unit. Further interpretation of the PSL verification unit is defined in IEEE Std 1850-2005.

Elaboration of a design hierarchy is completed as follows:

— The drivers identified during elaboration of process statements (see 14.5.5) are created.

— The initial transaction defined by the default value associated with each scalar signal driven by a process statement is inserted into the corresponding driver.

During elaboration of a design hierarchy, if an external name or alias of an external name appears in a declaration or statement being elaborated, then in the following cases, the declaration of the object denoted by the external name or alias shall have been previously elaborated:

— If the external name or alias is a primary or a prefix of a primary in an expression that is evaluated during elaboration of the design hierarchy, when the primary is read during evaluation of the expression.

— If the external name or alias, or a name in which the external name or alias is a prefix, is associated as an actual in an association element in a port map aspect, when the association element is elaborated.

NOTE—Since elaboration of declarations and statements occurs in the order of their appearance in a description, prior elaboration of an object denoted by an external name may be ensured by an appropriate ordering of the declarations and statements in the description.

Examples:

```
--  In the following example, because of the dependencies between
--  the packages, the elaboration of either package body shall
--  follow the elaboration of both package declarations.
```



```
package P1 is
   constant C1: INTEGER := 42;
   constant C2: INTEGER;
end package P1;

package P2 is
   constant C1: INTEGER := 17;
   constant C2: INTEGER;
end package P2;

package body P1 is
   constant C2: INTEGER := Work.P2.C1;
end package body P1;

package body P2 is
   constant C2: INTEGER := Work.P1.C1;
end package body P2;

--  If a design hierarchy is described by the following design entity:

entity E is end;

architecture A of E is
   component comp
      port (...);
   end component;
begin
   C: comp port map (...);
   B: block
      ...
   begin
      ...
   end block B;
end architecture A;

--  then its architecture contains the following implicit configuration
--  specification at the end of its declarative part:

for C: comp use configuration anonymous;

--  and the following configuration declaration is assumed to exist
--  when E(A) is elaborated:

configuration anonymous of L.E is    --  L is the library in which
                                     --  E(A) is found.
   for A                             --  The most recently analyzed
                                     --  architecture of L.E.
   end for;
end configuration anonymous;

--  In the following example, each appearance of an external name is
--  legal or illegal as noted.
```

```
entity TOP is
end entity TOP;

architecture ARCH of TOP is
   signal S1, S2, S3: BIT;
   alias DONE_SIG is <<signal .TOP.DUT.DONE: BIT>>;  -- Legal
   constant DATA_WIDTH: INTEGER
                        := <<signal .TOP.DUT.DATA: BIT_VECTOR>>'LENGTH;
      --  Illegal, because .TOP.DUT.DATA has not yet been elaborated
      --  when the expression is evaluated
begin
   P1: process ( DONE_SIG ) is  -- Legal
   begin
      if DONE_SIG then  -- Legal
         ...;
      end if;
   end process P1;
   MONITOR: entity WORK.MY_MONITOR port map (DONE_SIG);
      --  Illegal, because .TOP.DUT.DONE has not yet been elaborated
      --  when the association element is elaborated
   DUT: entity WORK.MY_DESIGN port map (s1, S2, S3);
   MONITOR2: entity WORK.MY_MONITOR port map (DONE_SIG);
      -- Legal, because .TOP.DUT.DONE has now been elaborated
   B1: block
      constant DATA_WIDTH: INTEGER
         := <<signal .TOP.DUT.DATA: BIT_VECTOR>>'LENGTH
         -- Legal, because .TOP.DUT.DATA has now been elaborated
   begin
   end block B1;
   B2: block
      constant C0: INTEGER := 6;
      constant C1: INTEGER := <<constant .TOP.B3.C2: INTEGER>>;
         -- Illegal, because .TOP.B3.C2 has not yet been elaborated
   begin
   end block B2;
   B3: block
      constant C2: INTEGER
                   := <<constant .TOP.B2.C0: INTEGER>>;  -- Legal
   begin
   end block B3;
   --  Together, B2 and B3 are illegal, because they cannot be ordered
   --  so that the objects are elaborated in the order .TOP.B2.C0,
   --  then .TOP.B3.C2, and finally .TOP.B2.C1.
end architecture ARCH;
```

14.3 Elaboration of a block, package, or subprogram header

14.3.1 General

Elaboration of a block header consists of the elaboration of the generic clause, the generic map aspect, the port clause, and the port map aspect. Similarly, elaboration of a package header consists of the elaboration of the generic clause and the generic map aspect; and elaboration of a subprogram header consists of the

elaboration of the generic clause equivalent to the generic list of the subprogram header and the generic map aspect.

14.3.2 Generic clause

Elaboration of a generic clause consists of the elaboration of each of the equivalent single generic declarations contained in the clause, in the order given. The elaboration of a generic declaration establishes that the generic can subsequently be referenced.

14.3.3 Generic map aspect

14.3.3.1 General

Elaboration of a generic map aspect consists of elaborating the generic association list. The generic association list contains an implicit association element for each generic constant that is not explicitly associated with an actual or that is associated with the reserved word **open**; the actual part of such an implicit association element is the default expression appearing in the declaration of that generic constant. Similarly, the generic association list contains an implicit association element for each generic subprogram that is not explicitly associated with an actual or that is associated with the reserved word **open**; the actual part of such an implicit association element is determined by the interface subprogram default as described in 6.5.6.2. The generic association list also contains implicit association elements for the predefined equality (=) operator and inequality (/=) operators of each generic type; the actual part of such an implicit association element is the name of the predefined equality operator or inequality operator for the base type of the subtype indication in the actual part of the association element corresponding to the generic type.

Elaboration of a generic association list consists of the elaboration of the generic association element or elements in the association list associated with each generic declaration, in the order given by the generic declarations in the generic clause.

14.3.3.2 Association elements for generic constants

Elaboration of the generic association elements associated with a generic constant declaration proceeds as follows:

a) The subtype indication of the corresponding generic declaration is elaborated.

b) The formal part or parts of the generic association elements corresponding to the generic declaration are elaborated.

c) If the type of the generic constant is an array type or contains a subelement that is of an array type, the rules of 5.3.2.2 are applied to determine the index ranges.

d) The generic constant is created.

The generic constant or subelement or slice thereof designated by each formal part is then initialized with the value resulting from the evaluation of the corresponding actual part. It is an error if the value of the actual does not belong to the subtype denoted by the subtype indication of the formal. If the subtype denoted by the subtype indication of the declaration of the formal is a composite subtype, then an implicit subtype conversion is performed prior to this check. It is also an error if the type of the formal is an array type and the value of each element of the actual does not belong to the element subtype of the formal.

14.3.3.3 Association elements for generic types

Elaboration of the generic association element associated with a generic type declaration involves the elaboration of the subtype indication in the actual part followed by creating the generic type and defining it to denote the subtype resulting from elaboration of the actual part.

14.3.3.4 Association elements for generic subprograms

Elaboration of the generic association element associated with a generic subprogram declaration proceeds as follows:

a) The parameter list of the formal generic subprogram declaration is elaborated. This involves the elaboration of the subtype indication of each interface element to determine the subtype of each formal parameter of the formal generic subprogram.

b) The generic subprogram is then defined to denote the subprogram denoted by the subprogram name in the actual part.

14.3.3.5 Association elements for generic packages

For a generic association element associated with a generic package declaration, if the generic package declaration contains an interface package generic map aspect in the form that includes the box (<>) symbol, elaboration of the generic association element involves defining the generic package to denote the instantiated package denoted by the instantiated package name in the actual part. Otherwise, elaboration of the generic association element proceeds as follows:

a) An implicit package header formed from the generic clause of the uninstantiated package named in the formal package declaration and the generic map aspect (whether explicit or implicit, see 6.5.5) of the interface package generic map aspect is elaborated.

b) A check is made that the generic map aspect of the package instantiation declaration that declares the instantiated package denoted by the instantiated package name in the actual part matches the elaborated generic map aspect of the implicit package header.

c) The generic package is defined to denote the instantiated package denoted by the instantiated package name in the actual part.

14.3.4 Port clause

Elaboration of a port clause consists of the elaboration of each of the equivalent single port declarations contained in the clause, in the order given. The elaboration of a port declaration establishes that the port can subsequently be referenced.

14.3.5 Port map aspect

Elaboration of a port map aspect consists of elaborating the port association list.

Elaboration of a port association list consists of the elaboration of the port association element or elements in the association list associated with each port declaration. If the actual in a port association element is an expression that is not globally static, or if the actual part includes the reserved word **inertial**, then elaboration of the port association element first consists of constructing and elaborating the equivalent anonymous signal declaration, concurrent signal assignment statement, and port association element (see 6.5.6.3); the port or subelement or slice thereof designated by the formal part is then associated with the anonymous signal.

Elaboration of the port association elements associated with a port declaration proceeds as follows:

a) The subtype indication of the corresponding port declaration is elaborated.

b) The formal part or parts of the port association elements corresponding to the port declaration are elaborated.

c) If the type of the port is an array type or contains a subelement that is of an array type, the rules of 5.3.2.2 are applied to determine the index ranges.

d) For each port association element associated with the port declaration, if the actual is not the reserved word **open**, the port or subelement or slice thereof designated by the formal part is then associated with the signal or expression designated by the actual part. This association involves a check that the restriction on port associations (see 6.5.6.3) are met. It is an error if this check fails.

If a given port is a port of mode **in** whose declaration includes a default expression, and if no association element associates a signal or expression with that port, then the default expression is evaluated and the effective and driving value of the port is set to the value of the default expression. Similarly, if a given port of mode **in** is associated with an expression that is globally static and the reserved word **inertial** does not appear in the actual part of the association element, that expression is evaluated and the effective and driving value of the port is set to the value of the expression. In the event that the value of a port is derived from an expression in either fashion, references to the predefined attributes 'DELAYED, 'STABLE, 'QUIET, 'EVENT, 'ACTIVE, 'LAST_EVENT, 'LAST_ACTIVE, 'LAST_VALUE, 'DRIVING, and 'DRIVING_VALUE of the port return values indicating that the port has the given driving value with no activity at any time (see 14.7.4).

If an actual signal is associated with a port of mode **in** or **inout**, and if the type of the formal is a scalar type, then it is an error if (after applying any conversion function or type conversion expression present in the actual part) the subtype of the actual is not compatible with the subtype of the formal. If an actual expression is associated with a formal port (of mode **in**), and if the type of the formal is a scalar type, then it is an error if the value of the expression does not belong to the subtype denoted by the subtype indication of the declaration of the formal.

Similarly, if an actual signal is associated with a port of mode **out**, **inout**, or **buffer**, and if the type of the actual is a scalar type, then it is an error if (after applying any conversion function or type conversion expression present in the formal part) the subtype of the formal is not compatible with the subtype of the actual.

If an actual signal or expression is associated with a formal port, and if the formal is of a composite subtype, then it is an error if the actual does not contain a matching element for each element of the formal. This check is made after applying the rules of 5.3.2.2 and, in the case of an actual signal, after applying any conversion function or type conversion that is present in the actual part. It is also an error if the mode of the formal is **in** or **inout** and the value of each element of the actual (after applying any conversion function or type conversion present in the actual part) does not belong to the corresponding element subtype of the formal. If the formal port is of mode **out**, **inout**, or **buffer**, it is also an error if the value of each element of the formal (after applying any conversion function or type conversion present in the formal part) does not belong to the corresponding element subtype of the actual.

14.4 Elaboration of a declarative part

14.4.1 General

The elaboration of a declarative part consists of the elaboration of the declarative items, if any, in the order in which they are given in the declarative part. This rule holds for all declarative parts, with the following three exceptions:

a) The entity declarative part of a design entity whose corresponding architecture is decorated with the 'FOREIGN attribute defined in package STANDARD (see 7.2 and 16.3) and for which the value of the attribute is not of the form described in 20.2.4.

b) The architecture declarative part of a design entity whose architecture is decorated with the 'FOREIGN attribute defined in package STANDARD and for which the value of the attribute is not of the form described in 20.2.4.

c) A subprogram declarative part whose subprogram is decorated with the 'FOREIGN attribute defined in package STANDARD.

For these cases, the declarative items are not elaborated; instead, the design entity or subprogram is subject to implementation-dependent elaboration.

In certain cases, the elaboration of a declarative item involves the evaluation of expressions that appear within the declarative item. The value of any object denoted by a primary in such an expression shall be defined at the time the primary is read (see 6.5.2). In addition, if a primary in such an expression is a function call, then the value of any object denoted by or appearing as a part of an actual designator in the function call shall be defined at the time the expression is evaluated. Additionally, it is an error if a primary that denotes a shared variable, or a method of the protected type of a shared variable, is evaluated during the elaboration of a declarative item. During static elaboration, the function STD.STANDARD.NOW (see 16.3) returns the value 0 ns.

NOTE 1—It is a consequence of this rule that the name of a signal declared within a block cannot be referenced in expressions appearing in declarative items within that block, an inner block, or process statement; nor can it be passed as a parameter to a function called during the elaboration of the block. These restrictions exist because the value of a signal is not defined until after the design hierarchy is elaborated. However, a signal parameter name may be used within expressions in declarative items within a subprogram declarative part, provided that the subprogram is only called after simulation begins, because the value of every signal will be defined by that time.

NOTE 2—A function called in an expression evaluated during elaboration of a declarative item may be a foreign function.

14.4.2 Elaboration of a declaration

14.4.2.1 General

Elaboration of a declaration has the effect of creating the declared item.

For each declaration, the language rules (in particular scope and visibility rules) are such that it is either impossible or illegal to use a given item before the elaboration of its corresponding declaration. For example, it is not possible to use the name of a type for an object declaration before the corresponding type declaration is elaborated. Similarly, it is illegal to call a subprogram before its corresponding body is elaborated.

Rules for creation of PSL declarations are defined in IEEE Std 1850-2005.

14.4.2.2 Subprogram declarations, bodies, and instantiations

Elaboration of a subprogram declaration, other than a subprogram declaration that defines an uninstantiated subprogram, involves the elaboration of the subprogram header, if present, followed by the elaboration of the parameter interface list of the subprogram declaration; the latter in turn involves the elaboration of the subtype indication of each interface element to determine the subtype of each formal parameter of the subprogram. Elaboration of an uninstantiated subprogram declaration simply establishes that the name of the subprogram may be referenced subsequently in subprogram instantiation declarations.

Elaboration of a subprogram body, other than the subprogram body of an uninstantiated subprogram, has no effect other than to establish that the body can, from then on, be used for the execution of calls of the subprogram. Elaboration of a subprogram body of an uninstantiated subprogram has no effect.

Elaboration of a subprogram instantiation declaration consists of elaboration of the equivalent generic-mapped subprogram declaration, followed by elaboration of the corresponding equivalent generic-mapped subprogram body (see 4.4). If the subprogram instantiation declaration occurs immediately within an enclosing package declaration, elaboration of the equivalent generic-mapped subprogram body occurs as

part of elaboration of the body, whether explicit or implicit, of the enclosing package. Similarly, if the subprogram instantiation declaration occurs immediately within an enclosing protected type declaration, elaboration of the equivalent generic-mapped subprogram body occurs as part of elaboration of the protected type body.

14.4.2.3 Type declarations

Elaboration of a type declaration generally consists of the elaboration of the definition of the type and the creation of that type. For a constrained type declaration that declares a partially or fully constrained composite subtype, however, elaboration consists of the elaboration of the equivalent anonymous unconstrained type followed by the elaboration of the named subtype of that unconstrained type.

Elaboration of an enumeration type definition has no effect other than the creation of the corresponding type.

Elaboration of an integer, floating-point, or physical type definition consists of the elaboration of the corresponding range constraint. For a physical type definition, each unit declaration in the definition is also elaborated. Elaboration of a physical unit declaration has no effect other than to create the unit defined by the unit declaration.

Elaboration of an unbounded array type definition that defines an unconstrained array type consists of the elaboration of the element subtype indication of the array type.

Elaboration of a record type definition consists of the elaboration of the equivalent single element declarations in the given order. Elaboration of an element declaration consists of elaboration of the element subtype indication.

Elaboration of an access type definition consists of the elaboration of the corresponding subtype indication.

Elaboration of a protected type definition consists of the elaboration, in the order given, of each of the declarations occurring immediately within the protected type definition.

Elaboration of a protected type body has no effect other than to establish that the body, from then on, can be used during the elaboration of objects of the given protected type.

14.4.2.4 Subtype declarations

Elaboration of a subtype declaration consists of the elaboration of the subtype indication. The elaboration of a subtype indication creates a subtype. If the subtype does not include a constraint, then the subtype is the same as that denoted by the type mark. The elaboration of a subtype indication that includes a constraint proceeds as follows:

a) The constraint is first elaborated.

b) A check is then made that the constraint is compatible with the type or subtype denoted by the type mark (see 5.2.1, 5.3.2.2, and 5.3.3).

Elaboration of a range constraint consists of the evaluation of the range. The evaluation of a range defines the bounds and direction of the range. Elaboration of an index constraint consists of the elaboration of each of the discrete ranges in the index constraint in some order that is not defined by the language. Elaboration of an array constraint consists of the elaboration of the index constraint, if present, and the elaboration of the array element constraint, if present. The order of elaboration of the index constraint and the array element constraint, if both are present, is not defined by the language. Elaboration of a record constraint consists of the elaboration of each of the record element constraints in the record constraint in some order that is not defined by the language.

14.4.2.5 Object declarations

The rules of this subclause apply only to explicitly declared objects (see 6.4.2.1). Generic declarations, port declarations, and other interface declarations are elaborated as described in 14.3.2 through 14.3.5 and 14.6.

Elaboration of an object declaration that declares an object other than a file object or an object of a protected type proceeds as follows:

a) The subtype indication is first elaborated; this establishes the subtype of the object.

b) If the object declaration includes an explicit initialization expression, then the initial value of the object is obtained by evaluating the expression. It is an error if the value of the expression does not belong to the subtype of the object; if the object is a composite object, then an implicit subtype conversion is first performed on the value unless the object is a constant whose subtype indication denotes an unconstrained type. Otherwise, any implicit initial value for the object is determined.

c) The object is created.

d) Any initial value is assigned to the object.

The initialization of such an object (either the declared object or one of its subelements) involves a check that the initial value belongs to the subtype of the object. For a composite object declared by an object declaration, an implicit subtype conversion is first applied as for an assignment statement, unless the object is a constant whose subtype is an unconstrained type.

The elaboration of a file object declaration consists of the elaboration of the subtype indication followed by the creation of the object. If the file object declaration contains file open information, then the implicit call to FILE_OPEN is then executed (see 6.4.2.5).

The elaboration of an object of a protected type consists of the elaboration of the subtype indication, followed by creation of the object. Creation of the object consists of elaborating, in the order given, each of the declarative items in the protected type body.

NOTE 1—The expression initializing a constant object need not be a static expression.

NOTE 2—Each object whose type is a protected type involves creation of separate instances of the objects declared by object declarations within the protected type body.

14.4.2.6 Alias declarations

Elaboration of an alias declaration consists of the elaboration of the subtype indication to establish the subtype associated with the alias, followed by the creation of the alias as an alternative name for the named entity. The creation of an alias for a composite object involves a check that the subtype associated with the alias includes a matching element for each element of the named object. It is an error if this check fails.

14.4.2.7 Attribute declarations

Elaboration of an attribute declaration has no effect other than to create a template for defining attributes of items.

14.4.2.8 Component declarations

Elaboration of a component declaration has no effect other than to create a template for instantiating component instances.

14.4.2.9 Packages

Elaboration of a package declaration, other than a package declaration that defines an uninstantiated package, consists of the elaboration of the package header, if present, followed by the elaboration of the declarative part of the package declaration. Elaboration of a package body, other than a package body of an uninstantiated package, consists of the elaboration of the declarative part of the package body. Elaboration of an uninstantiated package declaration simply establishes that the name of the package may be referenced subsequently in package instantiation declarations. Elaboration of a package body of an uninstantiated package has no effect.

Elaboration of a package instantiation declaration consists of elaboration of the equivalent generic-mapped package declaration, followed by elaboration of the corresponding equivalent generic-mapped package body, if such a package body is defined (see 4.9). If the package instantiation declaration occurs immediately within an enclosing package declaration and the uninstantiated package has a package body, elaboration of the equivalent generic-mapped package body occurs as part of elaboration of the body, whether explicit or implicit, of the enclosing package.

14.4.3 Elaboration of a specification

14.4.3.1 General

Elaboration of a specification has the effect of associating additional information with a previously declared item.

14.4.3.2 Attribute specifications

Elaboration of an attribute specification proceeds as follows:

a) The entity specification is elaborated in order to determine which items are affected by the attribute specification.
b) The expression is evaluated to determine the value of the attribute. It is an error if the value of the expression does not belong to the subtype of the attribute; if the attribute is of a composite type, then an implicit subtype conversion is first performed on the value, unless the subtype indication of the attribute denotes an unconstrained type.
c) A new instance of the designated attribute is created and associated with each of the affected items.
d) Each new attribute instance is assigned the value of the expression.

The assignment of a value to an instance of a given attribute involves a check that the value belongs to the subtype of the designated attribute. For an attribute of a partially or fully constrained composite type, an implicit subtype conversion is first applied as for an assignment statement. No such conversion is necessary for an attribute of an unconstrained type; the constraints on the value determine the constraints on the attribute.

NOTE—The expression in an attribute specification need not be a static expression.

14.4.3.3 Configuration specifications

Elaboration of a configuration specification proceeds as follows:

a) The component specification is elaborated in order to determine which component instances are affected by the configuration specification.
b) The binding indication is elaborated to identify the design entity to which the affected component instances will be bound.
c) The binding information is associated with each affected component instance label for later use in instantiating those component instances.

As part of this elaboration process, a check is made that both the entity declaration and the corresponding architecture body implied by the binding indication exist within the specified library. It is an error if this check fails.

14.4.3.4 Disconnection specifications

Elaboration of a disconnection specification proceeds as follows:

a) The guarded signal specification is elaborated in order to identify the signals affected by the disconnection specification.

b) The time expression is evaluated to determine the disconnection time for drivers of the affected signals.

c) The disconnection time is associated with each affected signal for later use in constructing disconnection statements in the equivalent processes for guarded assignments to the affected signals.

14.5 Elaboration of a statement part

14.5.1 General

Concurrent statements appearing in the statement part of a block shall be elaborated before execution begins. Elaboration of the statement part of a block consists of the elaboration of each concurrent statement in the order given. This rule holds for all block statement parts except for those blocks equivalent to a design entity whose corresponding architecture is decorated with the 'FOREIGN attribute defined in package STANDARD (see 16.3).

For this case, there are two subcases:

— If the value of the attribute is of the form described in 20.2.4, the statements are not elaborated; instead, the elaboration function of the foreign model is invoked, as described in 20.4.1, at the point where elaboration of the statements of the block statement corresponding to the architecture body would otherwise occur.

— Otherwise, the statements are not elaborated; instead, the design entity is subject to implementation-dependent elaboration.

Rules for interpretation of PSL directives are defined in IEEE Std 1850-2005.

14.5.2 Block statements

Elaboration of a block statement consists of the elaboration of the block header, if present, followed by the elaboration of the block declarative part, followed by the elaboration of the block statement part.

Elaboration of a block statement may occur under the control of a configuration declaration. In particular, a block configuration, whether implicit or explicit, within a configuration declaration may supply a sequence of additional implicit configuration specifications to be applied during the elaboration of the corresponding block statement. If a block statement is being elaborated under the control of a configuration declaration, then the sequence of implicit configuration specifications supplied by the block configuration is elaborated as part of the block declarative part, following all other declarative items in that part.

The sequence of implicit configuration specifications supplied by a block configuration, whether implicit or explicit, consists of each of the configuration specifications implied by component configurations (see 3.4.3) occurring immediately within the block configuration, in the order in which the component configurations themselves appear.

14.5.3 Generate statements

Elaboration of a generate statement consists of the replacement of the generate statement with zero or more copies of a block statement whose declarative part consists of declarative items contained within the generate statement and whose statement part consists of concurrent statements contained within the generate statement. These block statements are said to be *represented* by the generate statement. Each block statement is then elaborated.

For a for generate statement, elaboration consists of the elaboration of the discrete range, followed by the generation of one block statement for each value in the range. The block statements all have the following form:

a) The label of the block statement is the same as the label of the for generate statement.

b) The block declarative part has, as its first item, a single constant declaration that declares a constant with the same simple name as that of the applicable generate parameter; the value of the constant is the value of the generate parameter for the generation of this particular block statement. The type of this declaration is determined by the base type of the discrete range of the generate parameter. The remainder of the block declarative part consists of a copy of the declarative items contained within the generate statement.

c) The block statement part consists of a copy of the concurrent statements contained within the generate statement.

For an if generate statement, elaboration consists of the evaluation, in succession, of the condition specified after **if** and any conditions specified after **elsif** (treating a final **else** as **elsif** TRUE **generate**) until one evaluates to TRUE or all conditions are evaluated and yield FALSE. If one condition evaluates to TRUE, then exactly one block statement is generated; otherwise, no block statement is generated. If generated, the block statement has the following form:

— The block label is the same as the label of the if generate statement.

— The block declarative part consists of a copy of the declarative items contained within the generate statement body following the condition that evaluated to TRUE. If the condition is preceded by an alternative label, the label is implicitly declared at the beginning of the block declarative part.

— The block statement part consists of a copy of the concurrent statements contained within the generate statement body following the condition that evaluated to TRUE.

For a case generate statement, elaboration consists of the evaluation of the expression followed by the generation of a block statement for the chosen alternative. A given case generate alternative is the chosen alternative if and only if the expression “E = V” evaluates to TRUE, where “E” is the expression, “V” is the value of one of the choices of the given case generate alternative (if a choice is a discrete range, then this latter condition is fulfilled when V is an element of the discrete range), and the operator “=” in the expression is the predefined “=” operator on the base type of E. The generate block statement has the following form:

— The block label is the same as the label of the case generate statement.

— The block declarative part consists of a copy of the declarative items contained within the generate statement body of the chosen alternative. If the choices of the chosen alternative are preceded by an alternative label, the label is implicitly declared at the beginning of the block declarative part.

— The block statement part consists of a copy of the concurrent statements contained within the generate statement body of the chosen alternative.

Examples:

```
--  The following generate statement:
```

```
LABL: for I in 1 to 2 generate
   signal s1: INTEGER;
begin
   s1 <= p1;
   Inst1: and_gate port map (s1, p2(I), p3);
end generate LABL;

--  is equivalent to the following two block statements:

LABL: block
   constant I: INTEGER := 1;
   signal s1: INTEGER;
begin
   s1 <= p1;
   Inst1: and_gate port map (s1, p2(I), p3);
end block LABL;

LABL: block
   constant I: INTEGER := 2;
   signal s1: INTEGER;
begin
   s1 <= p1;
   Inst1: and_gate port map (s1, p2(I), p3);
end block LABL;

--  The following generate statement:

LABL: if (g1 = g2) generate
   signal s1: INTEGER;
begin
   s1 <= p1;
   Inst1: and_gate port map (s1, p4, p3);
end generate LABL;

--  is equivalent to the following statement if g1 = g2;
--  otherwise, it is equivalent to no statement at all:

LABL: block
   signal s1: INTEGER;
begin
   s1 <= p1;
   Inst1: and_gate port map (s1, p4, p3);
end block LABL;
```

NOTE—The repetition of the block labels in the case of a for generate statement does not produce multiple declarations of the label on the generate statement. The multiple block statements represented by the generate statement constitute multiple references to the same implicitly declared label.

14.5.4 Component instantiation statements

Elaboration of a component instantiation statement that instantiates a component declaration has no effect unless the component instance is either fully bound to a design entity defined by an entity declaration and architecture body or bound to a configuration of such a design entity. If a component instance is so bound, then elaboration of the corresponding component instantiation statement consists of the elaboration of the implied block statement representing the component instance and (within that block) the implied block

statement representing the design entity to which the component instance is bound. The implied block statements are defined in 11.7.2.

Elaboration of a component instantiation statement whose instantiated unit denotes either a design entity or a configuration declaration consists of the elaboration of the implied block statement representing the component instantiation statement and (within that block) the implied block statement representing the design entity to which the component instance is bound. The implied block statements are defined in 11.7.3.

14.5.5 Other concurrent statements

All other concurrent statements are either process statements or are statements for which there is an equivalent process statement.

Elaboration of a process statement proceeds as follows:

a) The process declarative part is elaborated.
b) The drivers required by the process statement are identified.

Elaboration of all concurrent signal assignment statements and concurrent assertion statements consists of the construction of the equivalent process statement followed by the elaboration of the equivalent process statement.

14.6 Dynamic elaboration

The execution of certain constructs that involve sequential statements rather than concurrent statements also involves elaboration. Such elaboration occurs during the execution of the model.

There are three particular instances in which elaboration occurs dynamically during simulation. These are as follows:

a) Execution of a loop statement with a for iteration scheme involves the elaboration of the loop parameter specification prior to the execution of the statements enclosed by the loop (see 10.10). This elaboration creates the loop parameter and evaluates the discrete range.
b) Execution of a subprogram call involves the elaboration of the parameter association list. This involves the elaboration of the parameter association element or elements in the association list associated with each interface declaration. Elaboration of the parameter association elements associated with a formal parameter declaration proceeds as follows:
    1) The subtype indication of the corresponding formal parameter declaration is elaborated.
    2) The formal part or parts of the parameter association elements corresponding to the formal parameter declaration are elaborated.
    3) If the type of the formal parameter is an array type or contains a subelement that is of an array type, the rules of 5.3.2.2 are applied to determine the index ranges.
    4) For each parameter association element associated with the formal parameter declaration, the parameter or subelement or slice thereof designated by the formal part is then associated with the actual part.
    5) If the formal parameter is a variable of mode **out**, then the implicit initial value for the object is determined.

    Next, if the subprogram is a method of a protected type (see 5.6.2) or an implicitly declared file operation (see 5.5.2), the elaboration *blocks* (suspends execution while retaining all state), if necessary, until exclusive access to the object denoted by the prefix of the method or to the file object denoted by the file parameter of the file operation is secured. Finally, if the designator of the subprogram is not decorated with the 'FOREIGN attribute defined in package STANDARD, the declarative part of the corresponding subprogram body is elaborated and the sequence of statements

in the subprogram body is executed. If the designator of the subprogram is decorated with the 'FOREIGN attribute defined in package STANDARD, there are two cases:

— If the value of the attribute is of the form described in 20.2.4, the declarative part of the corresponding subprogram body is not elaborated nor is the sequence of statements in the subprogram body executed; instead, the execution function of the foreign model is invoked, as described in 20.2.4.

— Otherwise, the subprogram body is subject to implementation-dependent elaboration and execution.

c) Evaluation of an allocator that contains a subtype indication involves the elaboration of the subtype indication prior to the allocation of the created object.

NOTE 1—It is a consequence of these rules that declarative items appearing within the declarative part of a subprogram body are elaborated each time the corresponding subprogram is called; thus, successive elaborations of a given declarative item appearing in such a place may create items with different characteristics. For example, successive elaborations of the same subtype declaration appearing in a subprogram body may create subtypes with different constraints.

NOTE 2—If two or more processes access the same set of shared variables, livelock or deadlock may occur. That is, it may not be possible to ever grant exclusive access to the shared variable as outlined in the preceding item b). Implementations are allowed to, but not required to, detect and, if possible, resolve such conditions.

14.7 Execution of a model

14.7.1 General

The elaboration of a design hierarchy produces a *model* that can be executed in order to simulate the design represented by the model. Simulation involves the execution of user-defined processes that interact with each other and with the environment. Simulation also involves interpretation of PSL directives to verify the properties that they specify.

The *kernel process* is a conceptual representation of the agent that coordinates the activity of user-defined processes during a simulation. This agent causes the propagation of signal values to occur and causes the values of implicit signals (such as S'STABLE) to be updated. Furthermore, this process is responsible for detecting events that occur and for causing the appropriate processes to execute in response to those events.

For any given signal that is explicitly declared within a model, the kernel process contains variables representing the driving value and current value of that signal. Any evaluation of a name denoting a given signal retrieves the current value of the corresponding variable in the kernel process. During simulation, the kernel process updates these variables from time to time, based upon the current values of sources of the corresponding signal.

In addition, the kernel process contains a variable representing the current value of any implicitly declared GUARD signal resulting from the appearance of a guard condition on a given block statement. Furthermore, the kernel process contains both a driver for, and a variable representing the current value of, any signal S'STABLE(T), for any prefix S and any time T, that is referenced within the model; likewise, for any signal S'QUIET(T) or S'TRANSACTION.

14.7.2 Drivers

Every signal assignment statement in a process statement defines a set of *drivers* for certain scalar signals. There is a single driver for a given scalar signal S in a process statement, provided that there is at least one signal assignment statement in that process statement and that the longest static prefix of the target signal of that signal assignment statement denotes S or denotes a composite signal of which S is a subelement. Each such signal assignment statement is said to be *associated* with that driver. Execution of a signal assignment statement affects only the associated driver(s).

A driver for a scalar signal is represented by a *projected output waveform*. A projected output waveform consists of a sequence of one or more *transactions*, where each transaction is a pair consisting of a value component and a time component. For a given transaction, the value component represents a value that the driver of the signal is to assume at some point in time, and the time component specifies which point in time. These transactions are ordered with respect to their time components.

A driver always contains at least one transaction. The initial contents of a driver associated with a given signal are defined by the default value associated with the signal (see 6.4.2.3). The kernel process contains a variable representing the *current value* of the driver. The initial value of the variable is the value component of the initial transaction of the driver.

For any driver, if, as the result of the advance of time, the current time becomes equal to the time component of the second transaction of the driver, the first transaction is deleted from the projected output waveform, and what was the second transaction becomes the first transaction. Then, or if a force or deposit is scheduled for the driver, the variable containing the current value of the driver is updated as follows:

— If a force is scheduled for the driver, the driver becomes forced and the variable containing the current value of the driver is updated with the force value for the driver.
— If the driver is forced and no force is scheduled for the driver, the variable containing the current value of the driver is unchanged from its previous value.
— If a deposit is scheduled for the driver and the driver is not forced, the variable containing the current value of the driver is updated with the deposit value for the driver.
— Otherwise, the variable containing the current value of the driver is updated with the value component of the first transaction of the driver.

When this action occurs on a driver, any registered and enabled `vhpiCbTransaction` callbacks associated with the given driver are executed. Moreover, if the current value of the driver changes as a result of this action, any registered and enabled `vhpiCbValueChange` callbacks associated with the given driver are executed.

14.7.3 Propagation of signal values

14.7.3.1 General

As simulation time advances, the transactions in the projected output waveform of a given driver (see 14.7.2) will each, in succession, become the value of the driver. When a driver acquires a new value in this way or as a result of a force or deposit scheduled for the driver, regardless of whether the new value is different from the previous value, that driver is said to be *active* during that simulation cycle. For the purposes of defining driver activity, a driver acquiring a value from a null transaction is assumed to have acquired a new value. A signal is said to be *active* during a given simulation cycle if

— One of its sources is active.
— One of its subelements is active.
— The signal is named in the formal part of an association element in a port association list and the corresponding actual is active.
— The signal is a subelement of a resolved signal and the resolved signal is active.
— A force, a deposit, or a release is scheduled for the signal.
— The signal is a subelement of another signal for which a force or a deposit is scheduled.

If a signal of a given composite type has a source that is of a different type (and therefore a conversion function or type conversion appears in the corresponding association element), then each scalar subelement of that signal is considered to be active if the source itself is active. Similarly, if a port of a given composite type is associated with a signal that is of a different type (and therefore a conversion function or type

conversion appears in the corresponding association element), then each scalar subelement of that port is considered to be active if the actual signal itself is active.

In addition to the preceding information, an implicit signal is said to be active during a given simulation cycle if the kernel process updates that implicit signal within the given cycle.

If a signal is not active during a given simulation cycle, then the signal is said to be *quiet* during that simulation cycle.

The kernel process determines two values for certain signals during certain simulation cycles. The *driving value* of a given signal is the value that signal provides as a source of other signals. The *effective value* of a given signal is the value obtainable by evaluating a reference to the signal within an expression. The driving value and the effective value of a signal are not always the same, especially when resolution functions and conversion functions or type conversions are involved in the propagation of signal values.

NOTE 1—In a given simulation cycle, situations can occur where a subelement of a composite signal is quiet, and the signal itself is active.

NOTE 2—The rules concerning association of actuals with formals (see 6.5.7.1) imply that, if a composite signal is associated with a composite port of mode **out**, **inout**, or **buffer**, and if no conversion function or type conversion appears in either the actual or formal part of the association element, then each scalar subelement of the formal is a source of the matching subelement of the actual. In such a case, a given subelement of the actual will be active if and only if the matching subelement of the formal is active.

NOTE 3—A signal of kind **register** may be active even if its associated resolution function does not execute in the current simulation cycle if the values of all of its drivers are determined by the null transaction and at least one of its drivers is also active.

14.7.3.2 Driving values

A *basic signal* is a signal that has all of the following properties:

- It is either a scalar signal or a resolved signal (see 6.4.2.3).
- It is not a subelement of a resolved signal.
- Is not an implicit signal of the form S'STABLE(T), S'QUIET(T), or S'TRANSACTION (see 16.2).
- It is not an implicit signal GUARD (see 11.2).

Basic signals are those that determine the driving values for all other signals.

The driving value of any signal S is determined by the following steps:

a) If a driving-value release is scheduled for S or for a signal of which S is a subelement, S becomes driving-value released, that is, no longer driving-value forced. Proceed to step b).

b) If a driving-value force is scheduled for S or for a signal of which S is a subelement, S becomes driving-value forced and the driving value of S is the driving force value for S or the element of the driving force value for the signal of which S is a subelement, as appropriate; no further steps are required. Otherwise, proceed to step c).

c) If S is driving-value forced, the driving value of S is unchanged from its previous value; no further steps are required. Otherwise, proceed to step d).

d) If a driving-value deposit is scheduled for S or for a signal of which S is a subelement, the driving value of S is the driving deposit value for S or the element of the driving deposit value for the signal of which S is a subelement, as appropriate; no further steps are required. Otherwise, proceed to step e) or f), as appropriate.

e) If S is a basic signal:

   — If S has no source, then the driving value of S is given by the default value associated with S (see 6.4.2.3).

— If S has one source that is a driver and S is not a resolved signal (see 6.4.2.3), then the driving value of S is the current value of that driver.

— If S has one source that is a port and S is not a resolved signal, then the driving value of S is the driving value of the formal part of the association element that associates S with that port (see 6.5.7.1). The driving value of a formal part is obtained by evaluating the formal part as follows: If no conversion function or type conversion is present in the formal part, then the driving value of the formal part is the driving value of the signal denoted by the formal designator. Otherwise, the driving value of the formal part is the value obtained by applying either the conversion function or type conversion (whichever is contained in the formal part) to the driving value of the signal denoted by the formal designator.

— If S is a resolved signal and has one or more sources, then the driving values of the sources of S are examined. It is an error if any of these driving values is a composite where one or more subelement values are determined by the null transaction (see 10.5.2.2) and one or more subelement values are not determined by the null transaction. If S is of signal kind **register** and all the sources of S have values determined by the null transaction, then the driving value of S is unchanged from its previous value. Otherwise, the driving value of S is obtained by executing the resolution function associated with S, where that function is called with an input parameter consisting of the concatenation of the driving values of the sources of S, with the exception of the value of any source of S whose current value is determined by the null transaction.

f) If S is not a basic signal:

— If S is a subelement of a resolved signal R, the driving value of S is the corresponding subelement value of the driving value of R.

— Otherwise (S is a nonresolved, composite signal), the driving value of S is equal to the aggregate of the driving values of each of the basic signals that are the subelements of S.

NOTE 1—The algorithm for computing the driving value of a scalar signal S is recursive. For example, if S is a local signal appearing as an actual in a port association list whose formal is of mode **out** or **inout**, the driving value of S can only be obtained after the driving value of the corresponding formal part is computed. This computation may involve multiple executions of the preceding algorithm.

NOTE 2—The definition of the driving value of a basic signal exhausts all cases, with the exception of a non-resolved signal with more than one source. This condition is defined as an error in 6.4.2.3.

NOTE 3—The driving value of a port that has no source is the default value of the port (see 6.5.2).

14.7.3.3 Effective values

For a scalar signal S, the *effective value* of S is determined by the following steps:

a) If an effective-value release is scheduled for S or for a signal of which S is a subelement, S becomes effective-value released, that is, no longer effective-value forced. Proceed to step b).

b) If an effective-value force is scheduled for S or for a signal of which S is a subelement, S becomes effective-value forced and the effective value of S is the effective force value for S or the element of the effective force value for the signal of which S is a subelement, as appropriate; no further steps are required. Otherwise, proceed to step c).

c) If S is effective-value forced, the effective value of S is unchanged from its previous value; no further steps are required. Otherwise, proceed to step d).

d) If an effective-value deposit is scheduled for S or for a signal of which S is a subelement, the effective value of S is the effective deposit value for S or the element of the effective deposit value for the signal of which S is a subelement, as appropriate; no further steps are required. Otherwise, proceed to step e).

e) The effective value of S is then determined as follows:

— If S is a signal declared by a signal declaration, a port of mode **out** or **buffer**, or an unconnected port of mode **inout**, then the effective value of S is the same as the driving value of S.

— If S is a connected port of mode **in** or **inout**, then the effective value of S is the same as the effective value of the actual part of the association element that associates an actual with S (see 6.5.7.1). The effective value of an actual part is obtained by evaluating the actual part, using the effective value of the signal denoted by the actual designator in place of the actual designator.

— If S is an unconnected port of mode **in**, the effective value of S is given by the default value associated with S (see 6.4.2.3).

For a composite signal R, the effective value of R is the aggregate of the effective values of each of the subelements of R.

NOTE 1—The algorithm for computing the effective value of a signal S is recursive. For example, if a formal port S of mode **in** corresponds to an actual A, the effective value of A shall be computed before the effective value of S can be computed. The actual A may itself appear as a formal port in a port association list.

NOTE 2—No effective value is specified for **linkage** ports, since these ports cannot be read.

14.7.3.4 Signal update

For a scalar signal S, both the driving and effective values shall belong to the subtype of the signal. For a composite signal R, an implicit subtype conversion is performed to the subtype of R; for each element of R, there shall be a matching element in both the driving and the effective value, and vice versa.

In order to update a signal during a given simulation cycle, the kernel process first determines the driving and effective values of that signal. The kernel process then updates the variable containing the driving value with the newly determined driving value. The kernel also updates the variable containing the current value of the signal with the newly determined effective value, as follows:

a) If S is a scalar signal, the effective value of S is used to update the current value of S. A check is made that the effective value of S belongs to the subtype of S. An error occurs if this subtype check fails. Finally, the effective value of S is assigned to the variable representing the current value of the signal.

b) If S is a composite signal (including a slice of an array), the effective value of S is implicitly converted to the subtype of S. The subtype conversion checks that for each element of S there is a matching element in the effective value and vice versa. An error occurs if this check fails. The result of this subtype conversion is then assigned to the variable representing the current value of S.

The current value of a signal of type T is said to *change* if and only if application of the predefined "=" operator for type T to the current value of the signal and the value of the signal prior to the update evaluates to FALSE. If updating a signal causes the current value of that signal to change, then an *event* is said to have occurred on the signal, unless the update occurs by application of the `vhpi_put_value` function with an update mode of `vhpiDeposit` or `vhpiForce` to an object that represents the signal. This definition applies to any updating of a signal, whether such updating occurs according to the preceding rules or according to the rules for updating implicit signals given in 14.7.4. The occurrence of an event will cause the resumption and subsequent execution of certain processes during the simulation cycle in which the event occurs, if and only if those processes are currently sensitive to the signal on which the event has occurred.

Each time a signal S is updated, any registered and enabled `vhpiCbTransaction` callbacks associated with S are executed. Each time there is an event on a signal S, any registered and enabled `vhpiCbValueChange` callbacks associated with S are executed.

A *net* is a collection of drivers, signals (including ports and implicit signals), conversion functions, and resolution functions that, taken together, determine the effective and driving values of every signal on the net.

For any signal that is part of a given net, the driving and effective values of the signal are determined and the variables containing the driving value and current value of that signal are updated as previously described in those simulation cycles in which any driver or signal on the net is active.

Implicit signals GUARD, S'STABLE(T), S'QUIET(T), and S'TRANSACTION, for any prefix S and any time T, are not updated according to the preceding rules; such signals are updated according to the rules described in 14.7.4.

NOTE 1—Overloading the operator "=" has no effect on the propagation of signal values.

NOTE 2—If a net includes an implicitly declared GUARD signal, the drivers of signals referred to in the corresponding guard condition determine the value of the GUARD signal. Hence, those drivers are part of the net, and when any of the drivers are active, the signals that are part of the net are updated.

14.7.4 Updating implicit signals

The kernel process updates the value of each implicit signal GUARD associated with a block statement that has a guard condition. Similarly, the kernel process updates the values of each implicit signal S'STABLE(T), S'QUIET(T), or S'TRANSACTION for any prefix S and any time T; this also involves updating the drivers of S'STABLE(T) and S'QUIET(T).

For any implicit signal GUARD, the current value of the signal is modified if and only if the corresponding guard condition contains a reference to a signal S and if S is active during the current simulation cycle. In such a case, the implicit signal GUARD is updated by evaluating the corresponding guard condition and assigning the result of that evaluation to the variable representing the current value of the signal. Whenever an implicit signal GUARD is updated, any registered and enabled `vhpiCbTransaction` callbacks associated with the given signal are executed.

For any implicit signal S'STABLE(T), the current value of the signal (and likewise the current state of the corresponding driver) is modified if and only if one of the following statements is true:

— An event has occurred on S in this simulation cycle.

— The driver of S'STABLE(T) is active.

If an event has occurred on signal S, then S'STABLE(T) is updated by assigning the value FALSE to the variable representing the current value of S'STABLE(T), and the driver of S'STABLE(T) is assigned the waveform TRUE **after** T. Otherwise, if the driver of S'STABLE(T) is active, then S'STABLE(T) is updated by assigning the current value of the driver to the variable representing the current value of S'STABLE(T). Otherwise, neither the variable nor the driver is modified. Whenever a signal of the form S'STABLE(T) is updated, any registered and enabled `vhpiCbTransaction` callbacks associated with the given signal are executed.

Similarly, for any implicit signal S'QUIET(T), the current value of the signal (and likewise the current state of the corresponding driver) is modified if and only if one of the following statements is true:

— S is active.

— The driver of S'QUIET(T) is active.

If signal S is active, then S'QUIET(T) is updated by assigning the value FALSE to the variable representing the current value of S'QUIET(T), and the driver of S'QUIET(T) is assigned the waveform TRUE **after** T. Otherwise, if the driver of S'QUIET(T) is active, then S'QUIET(T) is updated by assigning the current value of the driver to the variable representing the current value of S'QUIET(T). Otherwise, neither the variable nor the driver is modified. Whenever a signal of the form S'QUIET(T) is updated, any registered and enabled `vhpiCbTransaction` callbacks associated with the given signal are executed.

Finally, for any implicit signal S'TRANSACTION, the current value of the signal is modified if and only if S is active. If signal S is active, then S'TRANSACTION is updated by assigning the value of the expression (**not** S'TRANSACTION) to the variable representing the current value of S'TRANSACTION. At most one such assignment will occur during any given simulation cycle. Whenever a signal of the form S'TRANSACTION is updated, any registered and enabled `vhpiCbTransaction` callbacks associated with the given signal are executed.

For any implicit signal S'DELAYED(T), the signal is not updated by the kernel process. Instead, it is updated by constructing an equivalent process (see 16.2) and executing that process.

Each time there is an event on a signal S, where S is any one of

— An implicit signal GUARD

— P'STABLE(T), for any prefix P and any time T

— P'QUIET(T), for any prefix P and any time T

— P'TRANSACTION, for any prefix P

any registered and enabled `vhpiCbValueChange` callbacks associated with S are executed.

The current value of a given implicit signal denoted by R is said to *depend* upon the current value of another signal S if one of the following statements is true:

— R denotes an implicit GUARD signal and S is any other implicit signal named within the guard condition that defines the current value of R.

— R denotes an implicit signal S'STABLE(T).

— R denotes an implicit signal S'QUIET(T).

— R denotes an implicit signal S'TRANSACTION.

— R denotes an implicit signal S'DELAYED(T).

Similarly, the current value of a given interface signal denoted by R is said to *depend* upon the current value of an implicit signal S if R denotes a port of mode **in** and S is the actual associated with that port.

These rules define a partial ordering on all signals within a model. The updating of signals by the kernel process is guaranteed to proceed in such a manner that, if a given implicit signal R depends upon the current value of another signal S, or if a given interface signal R depends upon the value of an implicit signal S, then the current value of S will be updated during a particular simulation cycle prior to the updating of the current value of R.

NOTE—These rules imply that, if the driver of S'STABLE(T) is active, then the new current value of that driver is the value TRUE. Furthermore, these rules imply that, if an event occurs on S during a given simulation cycle, and if the driver of S'STABLE(T) becomes active during the same cycle, the variable representing the current value of S'STABLE(T) will be assigned the value FALSE, and the current value of the driver of S'STABLE(T) during the given cycle will never be assigned to that signal.

14.7.5 Model execution

14.7.5.1 General

The execution of a model consists of an initialization phase followed by the repetitive execution of process statements in the description of that model. Each such repetition is said to be a *simulation cycle*. In each cycle, the values of all signals in the description are computed. If as a result of this computation an event occurs on a given signal, process statements that are sensitive to that signal will resume and will be executed as part of the simulation cycle.

At certain stages during the initialization phase and each simulation cycle, the current time, T_c, and the time of the next simulation cycle, T_n, are calculated. T_n is calculated by setting it to the earliest of

a) TIME'HIGH,

b) The next time at which a driver or signal becomes active,

c) The next time at which a process resumes, or

d) The next time at which a registered and enabled `vhpiCbAfterDelay`, `vhpiCbRepAfterDelay`, `vhpiCbTimeOut`, or `vhpiCbRepTimeOut` callback is to occur.

If $T_n = T_c$, then the next simulation cycle (if any) will be a *delta cycle*.

14.7.5.2 Initialization

At the beginning of initialization, the current time, T_c, is assumed to be 0 ns.

The initialization phase consists of the following steps:

a) Each registered and enabled `vhpiCbStartofInitialization` callback is executed.

b) Each registered and enabled `vhpiCbStartOfNextCycle` and `vhpiCbRepStartOfNextCycle` callback is executed.

c) The signals in the model are updated as follows in an order such that if a given signal R depends upon the current value of another signal S, then the current value of S is updated prior to the updating of the current value of R:

   — The driving value and the effective value of each explicitly declared signal are computed, and the variables representing the driving value and current value of the signal are set to the driving value and effective value, respectively. The current value is assumed to have been the value of the signal for an infinite length of time prior to the start of simulation. If a force, deposit, or release was scheduled for any driver or signal, the force, deposit or release is no longer scheduled for the driver or signal.

   — The value of each implicit signal of the form S'STABLE(T) or S'QUIET(T) is set to TRUE. The value of each implicit signal of the form S'DELAYED(T) is set to the initial value of its prefix, S.

   — The value of each implicit GUARD signal is set to the result of evaluating the corresponding guard condition.

d) Any action required to give effect to a PSL directive is performed (see IEEE Std 1850-2005).

e) Each registered and enabled `vhpiCbStartOfProcesses` and `vhpiCbRepStartOfProcesses` callback is executed.

f) For each nonpostponed process P in the model, the following actions occur in the indicated order:

   1) The process executes until it suspends.

   2) Each registered and enabled `vhpiCbSuspend` callback associated with P is executed.

g) For each elaborated instance of a registered foreign architecture, the corresponding execution function is invoked.

h) Each registered and enabled `vhpiCbEndOfProcesses` and `vhpiCbRepEndOfProcesses` callback is executed.

i) Each registered and enabled `vhpiCbStartOfPostponed` and `vhpiCbRepStartOfPostponed` callback is executed.

j) For each postponed process P in the model, the following actions occur in the indicated order:

   1) The process executes until it suspends.

   2) Each registered and enabled `vhpiCbSuspend` callback associated with P is executed.

k) The time of the next simulation cycle (which in this case is the first simulation cycle), T_n, is calculated according to the rules of 14.7.5.1.

l) If the VHDL tool executing the initialization phase has requested a model save that has not yet been satisfied, the model is saved as described in 20.7.

m) Each registered and enabled `vhpiCbEndOfInitialization` callback is executed.

NOTE 1—The initial value of any implicit signal of the form S'TRANSACTION is not defined.

NOTE 2—Updating of explicit signals is described in 14.7.3; updating of implicit signals is described in 14.7.4.

NOTE 3—`vhpiCbResume` callbacks are not executed during initialization as processes do not resume during initialization.

14.7.5.3 Simulation cycle

A simulation cycle consists of the following steps:

a) The current time, T_c, is set equal to T_n. Simulation is complete when T_n = TIME'HIGH and there are no active drivers, process resumptions, or registered and enabled `vhpiCbAfterDelay`, `vhpiCbRepAfterDelay`, `vhpiCbTimeOut`, or `vhpiCbRepTimeOut` callbacks to occur at T_n.

b) The following actions occur in the indicated order:

1) If the current simulation cycle is not a delta cycle, each registered and enabled `vhpiCbNextTimeStep` and `vhpiCbRepNextTimeStep` callback is executed.

2) Each registered and enabled `vhpiCbStartOfNextCycle` and `vhpiCbRepStartOfNextCycle` callback is executed.

3) Each registered and enabled `vhpiCbAfterDelay` and `vhpiCbRepAfterDelay` callback is executed.

c) Each active driver in the model is updated. If a force or deposit was scheduled for any driver, the force or deposit is no longer scheduled for the driver.

d) Each signal on each net in the model that includes active drivers is updated in an order that is consistent with the dependency relation between signals (see 14.7.4). (Events may occur on signals as a result.) If a force, deposit, or release was scheduled for any signal, the force, deposit, or release is no longer scheduled for the signal.

e) Any action required to give effect to a PSL directive is performed (see IEEE Std 1850-2005).

f) The following actions occur in the indicated order:

1) Each registered and enabled `vhpiCbStartOfProcesses` and `vhpiCbRepStartOfProcesses` callback is executed. If an event has occurred on a signal S in this simulation cycle, then each registered and enabled `vhpiCbSensitivity` callback associated with S is executed.

2) For each process, P, if P is currently sensitive to a signal, S, and if an event has occurred on S in this simulation cycle, then P resumes.

3) Each registered and enabled `vhpiCbTimeOut` and `vhpiCbRepTimeOut` callback whose triggering condition is met is executed. For each nonpostponed process P that has resumed in the current simulation cycle, the following actions occur in the indicated order:

— Each registered and enabled `vhpiCbResume` callback associated with P is executed.

— The process executes until it suspends.

— Each registered and enabled `vhpiCbSuspend` callback associated with P is executed.

4) Each registered and enabled `vhpiCbEndOfProcesses` and `vhpiCbRepEndOfProcesses` callback is executed.

g) The time of the next simulation cycle, T_n, is calculated according to the rules of 14.7.5.1.

h) If the next simulation cycle will be a delta cycle, the remainder of step h) is skipped. Otherwise, the following actions occur in the indicated order:

   1) Each registered and enabled `vhpiCbLastKnownDeltaCycle` and `vhpiCbRepLastKnownDeltaCycle` callback is executed. T_n is recalculated according to the rules of 14.7.5.1.
   2) If the next simulation cycle will be a delta cycle, the remainder of step h) is skipped.
   3) Each registered and enabled `vhpiCbStartOfPostponed` and `vhpiCbRepStartOfPostponed` callback is executed.
   4) For each postponed process P, if P has resumed but has not been executed since its last resumption, the following actions occur in the indicated order:
      — Each registered and enabled `vhpiCbResume` callback associated with P is executed.
      — The process executes until it suspends.
      — Each registered and enabled `vhpiCbSuspend` callback associated with P is executed.
   5) T_n is recalculated according to the rules of 14.7.5.1.
   6) Each registered and enabled `vhpiCbEndOfTimeStep` and `vhpiCbRepEndOfTimeStep` callback is executed.
   7) If T_n = TIME'HIGH and there are no active drivers, process resumptions, or registered and enabled `vhpiCbAfterDelay`, `vhpiCbRepAfterDelay`, `vhpiCbTimeOut`, or `vhpiCbRepTimeOut` callbacks to occur at T_n, then each registered and enabled `vhpiCbQuiescence` callback is executed. T_n is recalculated according to the rules of 14.7.5.1.

   It is an error if the execution of any postponed process or any callback executed in substeps 3) through 7) of step h) causes a delta cycle to occur immediately after the current simulation cycle.

i) If the VHDL tool executing the simulation cycle has requested a model save that has not yet been satisfied, the model is saved as described in 20.7.

Immediately prior to the execution of the first simulation cycle, each registered and enabled `vhpiCbStartOfSimulation` callback is executed. Immediately subsequent to the execution of the final simulation cycle (i.e., when simulation is complete), each registered and enabled `vhpiCbEndOfSimulation` callback is executed.

NOTE 1—Updating of explicit signals is described in 14.7.3; updating of implicit signals is described in 14.7.4.

NOTE 2—When a process resumes, it is added to one of two sets of processes to be executed (the set of postponed processes and the set of nonpostponed processes). However, no process actually begins to execute until all signals have been updated and all executable processes for this simulation cycle have been identified. Nonpostponed processes are always executed during step f) of every simulation cycle, while postponed processes are executed during step h) of every simulation cycle that does not immediately precede a delta cycle.

NOTE 3—The `vhpiCbEndOfTimeStep` and `vhpiCbRepEndOfTimeStep` callbacks cannot cause activity or register callbacks that would result in a change to the time of the next simulation cycle, T_n (see 21.3.6.8).

15. Lexical elements

15.1 General

The text of a description consists of one or more design files. The text of a design file is a sequence of lexical elements, each composed of characters; the rules of composition are given in this clause.

15.2 Character set

The only characters allowed in the text of a VHDL description (except within comments—see 15.9, and within text treated specially due to the effect of tool directives—see 15.11) are the graphic characters and format effectors. Each graphic character corresponds to a unique code of the ISO eight-bit coded character set (ISO/IEC 8859-1:1998) and is represented (visually) by a graphical symbol.

basic_graphic_character ::=
    upper_case_letter | digit | special_character | space_character

graphic_character ::=
    basic_graphic_character | lower_case_letter | other_special_character

basic_character ::=
    basic_graphic_character | format_effector

The basic character set is sufficient for writing any description, other than a PSL declaration, a PSL directive, or a PSL verification unit. The characters included in each of the categories of basic graphic characters are defined as follows:

— Uppercase letters
— A B C D E F G H I J K L M N O P Q R S T U V W X Y Z À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î
— Ï Ð Ñ Ò Ó Ô Õ Ö Ø Ù Ú Û Ü Ý Þ
— Digits
— 0 1 2 3 4 5 6 7 8 9
— Special characters
— " # & ' ( ) * + , - . / : ; < = > ? @ [ ] _ ` |
— The space characters
— SPACE[7] NBSP[8]

Format effectors are the ISO/IEC (and ASCII) characters called horizontal tabulation, vertical tabulation, carriage return, line feed, and form feed.

The characters included in each of the remaining categories of graphic characters are defined as follows:

— Lowercase letters

  a b c d e f g h i j k l m n o p q r s t u v w x y z ß à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô
  õ ö ø ù ú û ü ý þ ÿ

— Other special characters

  ! $ % \ ^ { } ~ ¡ ¢ £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯ ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿ × ÷ - (soft hyphen)

[7]The visual representation of the space is the absence of a graphic symbol. It may be interpreted as a graphic character, a control character, or both.

[8]The visual representation of the nonbreaking space is the absence of a graphic symbol. It is used when a line break is to be prevented in the text as presented.

For each uppercase letter, there is a corresponding lowercase letter; and for each lowercase letter except ÿ and ß, there is a corresponding uppercase letter. The pairs of corresponding uppercase and lowercase letters are:

A a	B b	C c	D d	E e	F f	G g
H h	I i	J i	K k	L l	M m	N n
O o	P p	Q q	R r	S s	T t	U u
V v	W w	X x	Y y	Z z	À à	Á á
Â â	Ã ã	Ä ä	Å å	Æ æ	Ç ç	È è
É é	Ê ê	Ë ë	Ì ì	Í í	Î î	Ï ï
Ð ð	Ñ ñ	Ò ò	Ó ó	Ô ô	Õ õ	Ö ö
Ø ø	Ù ù	Ú ú	Û û	Ü ü	Ý ý	Þ þ

Within a PSL declaration, a PSL directive, or a PSL verification unit, certain of the other special characters are allowed (see 15.3 and IEEE Std 1850-2005).

NOTE 1—The font design of graphical symbols (for example, whether they are in italic or bold typeface) is not part of ISO/IEC 8859-1:1998.

NOTE 2—The meanings of the acronyms used in this subclause are as follows: ASCII stands for American Standard Code for Information Interchange, ISO stands for International Organization for Standardization.

NOTE 3—There are no uppercase equivalents for the characters ß and ÿ.

NOTE 4—The following names are used when referring to special characters:

Character	Name		
"	Quotation mark	£	Pound sign
#	Number sign	¤	Currency sign
&	Ampersand	¥	Yen sign
'	Apostrophe, tick	¦	Broken bar
(	Left parenthesis	§	Paragraph sign, clause sign
)	Right parenthesis	¨	Diaeresis
*	Asterisk, multiply	©	Copyright sign
+	Plus sign	ª	Feminine ordinal indicator
,	Comma	«	Left angle quotation mark
-	Hyphen, minus sign	¬	Not sign
.	Dot, point, period, full stop	-	Soft hyphen[a]
/	Slash, divide, solidus	®	Registered trade mark sign
:	Colon	¯	Macron
;	Semicolon	°	Ring above, degree sign
<	Less-than sign	±	Plus-minus sign

Character	Name		
=	Equals sign	²	Superscript two
>	Greater-than sign	³	Superscript three
_	Underline, low line	´	Acute accent
|	Vertical line, vertical bar	µ	Micro sign
!	Exclamation mark	¶	Pilcrow sign
$	Dollar sign	·	Middle dot
%	Percent sign	¸	Cedilla
?	Question mark	¹	Superscript one
@	Commercial at	º	Masculine ordinal indicator
[	Left square bracket	»	Right angle quotation mark
\	Backslash, reverse solidus	¼	Vulgar fraction one quarter
]	Right square bracket	½	Vulgar fraction one half
^	Circumflex accent	¾	Vulgar fraction three quarters
`	Grave accent	¿	Inverted question mark
{	Left curly bracket	×	Multiplication sign
}	Right curly bracket	÷	Division sign
~	Tilde		
¡	Inverted exclamation mark		
¢	Cent sign		

[a]The soft hyphen is a graphic character that is represented by a graphic symbol identical with, or similar to, that representing a hyphen, for use when a line break has been established within a word.

15.3 Lexical elements, separators, and delimiters

The text of each design unit, apart from text treated specially due to the effect of tool directives (see 15.11), is a sequence of separate lexical elements. Each lexical element is either a delimiter, an identifier (which may be a reserved word), an abstract literal, a character literal, a string literal, a bit string literal, a comment, a lexical element defined for a tool directive, or a lexical element defined in IEEE Std 1850-2005 for a PSL declaration, a PSL directive, or a PSL verification unit.

In some cases an explicit separator is required to separate adjacent lexical elements (namely when, without separation, interpretation as a single lexical element is possible). A separator is either a space character (SPACE or NBSP), a format effector, or the end of a line. A space character (SPACE or NBSP) is a separator except within an extended identifier, a comment, a string literal, a space character literal, or where defined to be part of a lexical element in a tool directive.

The end of a line is always a separator. The language does not define what causes the end of a line. However if, for a given implementation, the end of a line is signified by one or more characters, then these characters shall be format effectors other than horizontal tabulation. In any case, a sequence of one or more format effectors other than horizontal tabulation shall cause at least one end-of-line.

One or more separators are allowed between any two adjacent lexical elements, before the first of each design unit, or after the last lexical element of a design file. At least one separator is required between an identifier or an abstract literal and an adjacent identifier or abstract literal.

A delimiter is either one of the following special characters (in the basic character set):

& ' ( ) * + , - . / : ; < = > ` | [ ] ? @

or one of the following compound delimiters, each composed of two or more adjacent special characters:

=> ** := /= >= <= <> ?? ?= ?/= ?< ?<= ?> ?>= << >>

Each of the special characters listed for single character delimiters is a single delimiter except if this character is used as a character of a compound delimiter or as a character of an extended identifier, comment, string literal, character literal, or abstract literal.

The remaining forms of lexical elements are described in subclauses of this clause.

NOTE 1—Each lexical element shall fit on one line, since the end of a line is a separator. The quotation mark, number sign, and underline characters, likewise two adjacent hyphens, are not delimiters, but may form part of other lexical elements.

NOTE 2—The following names are used when referring to compound delimiters:

Delimiter	Name
=>	Arrow
**	Double star, exponentiate
:=	Variable assignment
/=	Inequality (pronounced “not equal”)
>=	Greater than or equal
<=	Less than or equal; signal assignment
<>	Box
??	Condition conversion
?=	Matching equality
?/=	Matching inequality
?<	Matching less than
?<=	Matching less than or equal
?>	Matching greater than
?>=	Matching greater than or equal
<<	Double less than
>>	Double greater than

NOTE 3—PSL macros and preprocessing directives can only be defined and used within PSL verification units. They cannot appear in PSL declarations or PSL directives embedded in other VHDL code, since they do not occur as part of the syntax of PSL declarations or PSL directives.

15.4 Identifiers

15.4.1 General

Identifiers are used as names and also as reserved words.

identifier ::= basic_identifier | extended_identifier

15.4.2 Basic identifiers

A basic identifier consists only of letters, digits, and underlines.

basic_identifier ::=
    letter { [underline] letter_or_digit }

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter

All characters of a basic identifier are significant, including any underline character inserted between a letter or digit and an adjacent letter or digit. Basic identifiers differing only in the use of corresponding uppercase and lowercase letters are considered the same.

Examples:

```
COUNT     X      c_out         FFT                 Decoder
VHSIC     X1     PageCount     STORE_NEXT_ITEM
```

NOTE—No space (SPACE or NBSP) is allowed within a basic identifier, since a space is a separator.

15.4.3 Extended identifiers

Extended identifiers may contain any graphic character.

extended_identifier ::=
    \ graphic_character { graphic_character } \

If a backslash is to be used as one of the graphic characters of an extended identifier, it shall be doubled. All characters of an extended identifier are significant (a doubled backslash counting as one character). Extended identifiers differing only in the use of corresponding uppercase and lowercase letters are distinct. Moreover, every extended identifier is distinct from any basic identifier.

Examples:

```
\BUS\      \bus\                  --  Two different identifiers,
                                  --  neither of which is
                                  --  the reserved word bus.
\a\\b\                            --  An identifier containing
                                  --  three characters.
VHDL       \VHDL\       \vhdl\    --  Three distinct identifiers.
```

15.5 Abstract literals

15.5.1 General

There are two classes of abstract literals: real literals and integer literals. A real literal is an abstract literal that includes a point; an integer literal is an abstract literal without a point. Real literals are the literals of the type *universal_real*. Integer literals are the literals of the type *universal_integer*.

abstract_literal ::= decimal_literal | based_literal

15.5.2 Decimal literals

A decimal literal is an abstract literal expressed in the conventional decimal notation (that is, the base is implicitly ten).

decimal_literal ::= integer [. integer] [exponent]

integer ::= digit { [underline] digit }

exponent ::= E [+] integer | E – integer

An underline character inserted between adjacent digits of a decimal literal does not affect the value of this abstract literal. The letter E of the exponent, if any, can be written either in lowercase or in uppercase, with the same meaning.

An exponent indicates the power of 10 by which the value of the decimal literal without the exponent is to be multiplied to obtain the value of the decimal literal with the exponent. An exponent for an integer literal shall not have a minus sign.

Examples:

```
12          0         1E6          123_456       --  Integer literals.
12.0        0.0       0.456        3.14159_26    --  Real literals.
1.34E-12    1.0E+6    6.023E+24                  --  Real literals
                                                 --  with exponents.
```

NOTE—Leading zeros are allowed. No space (SPACE or NBSP) is allowed in an abstract literal, not even between constituents of the exponent, since a space is a separator. A zero exponent is allowed for an integer literal.

15.5.3 Based literals

A based literal is an abstract literal expressed in a form that specifies the base explicitly. The base shall be at least two and at most sixteen.

based_literal ::=
    base # based_integer [. based_integer] # [exponent]

base ::= integer

based_integer ::=
    extended_digit { [underline] extended_digit }

extended_digit ::= digit | letter

An underline character inserted between adjacent digits of a based literal does not affect the value of this abstract literal. The base and the exponent, if any, are in decimal notation. The only letters allowed as extended digits are the letters A through F for the digits 10 through 15. A letter in a based literal (either an extended digit or the letter E of an exponent) can be written either in lowercase or in uppercase, with the same meaning.

The conventional meaning of based notation is assumed; in particular the value of each extended digit of a based literal shall be less than the base. An exponent indicates the power of the base by which the value of the based literal without the exponent is to be multiplied to obtain the value of the based literal with the exponent. An exponent for a based integer literal shall not have a minus sign.

Examples:

```
--  Integer literals of value 255:
    2#1111_1111#               16#FF#                 016#0FF#

--  Integer literals of value 224:
    16#E#E1                    2#1110_0000#

--  Real literals of value 4095.0:
    16#F.FF#E+2                2#1.1111_1111_111#E11
```

15.6 Character literals

A character literal is formed by enclosing one of the 191 graphic characters (including the space and nonbreaking space characters) between two apostrophe characters. A character literal has a value that belongs to a character type.

character_literal ::= ' graphic_character '

Examples:

```
'A'   '*'  '''   ' '
```

15.7 String literals

A string literal is formed by a sequence of graphic characters (possibly none) enclosed between two quotation marks used as string brackets.

string_literal ::= " { graphic_character } "

A string literal has a value that is a sequence of character values corresponding to the graphic characters of the string literal apart from the quotation mark itself. If a quotation mark value is to be represented in the sequence of character values, then a pair of adjacent quotation marks shall be written at the corresponding place within the string literal. (This means that a string literal that includes two adjacent quotation marks is never interpreted as two adjacent string literals.)

The length of a string literal is the number of character values in the sequence represented. (Each doubled quotation mark is counted as a single character.)

Examples:

```
"Setup time is too short"       --  An error message.
```

```
""                                  --  An empty string literal.
" "    "A"   """"                   --  Three string literals of length 1.

"Characters such as $, %, and } are allowed in string literals."
```

NOTE—A string literal shall fit on one line, since it is a lexical element (see 15.3). Longer sequences of graphic character values can be obtained by concatenation of string literals. The concatenation operation may also be used to obtain string literals containing nongraphic character values. The predefined type CHARACTER in package STANDARD specifies the enumeration literals denoting both graphic and nongraphic characters. Examples of such uses of concatenation are as follows:

```
"FIRST PART OF A SEQUENCE OF CHARACTERS " &
"THAT CONTINUES ON THE NEXT LINE"

"Sequence that includes the" & ACK & "control character"
```

15.8 Bit string literals

A bit string literal is formed by a sequence of characters (possibly none) enclosed between two quotation marks used as bit string brackets, preceded by a base specifier. The bit string literal may also be preceded by an integer specifying the length of the value represented by the bit string literal.

bit_string_literal ::= [integer] base_specifier " [bit_value] "

bit_value ::= graphic_character { [underline] graphic_character }

base_specifier ::= B | O | X | UB | UO | UX | SB | SO | SX | D

A graphic character in a bit string literal shall not be an underline character. An underline character inserted between adjacent graphic characters of a bit string literal does not affect the value of this literal.

If the base specifier is B, UB or SB, the digits 0 and 1 in the bit value are interpreted as extended digits, and all other graphic characters are not interpreted as extended digits. If the base specifier is O, UO, or SO, the digits 0 through 7 in the bit value are interpreted as extended digits, and all other graphic characters are not interpreted as extended digits. If the base specifier is X, UX or SX, all digits together with the letters A through F in the bit value are interpreted as extended digits. If the base specifier is D, all of the graphic characters in the bit value (not counting underline characters) shall be digits. An extended digit and the base specifier in a bit string literal can be written either in lowercase or in uppercase, with the same meaning.

A bit string literal has a value that is a string literal. The string literal is formed from the bit value by first obtaining a *simplified bit value*, consisting of the bit value with underline characters removed, and then obtaining an *expanded bit value*. Finally, the string literal value is obtained by adjusting the expanded bit value, if required.

If the base specifier is B, UB or SB, the expanded bit value is the simplified bit value itself. If the base specifier is O, UO, or SO (respectively X, UX, or SX), the expanded bit value is the string obtained by replacing each character of the simplified bit value by a sequence of three (respectively four) characters. For a character in the simplified bit value that is interpreted as an extended digit, the replacement sequence is as follows:

Extended digit	Replacement when the base specifier is O, UO, or SO	Replacement when the base specifier is X, UX, or SX
0	000	0000
1	001	0001
2	010	0010
3	011	0011
4	100	0100
5	101	0101
6	110	0110
7	111	0111
8		1000
9		1001
A		1010
B		1011
C		1100
D		1101
E		1110
F		1111

For a character in the simplified value that is not interpreted as an extended digit, each character in the replacement sequence is the same as the character replaced.

If the base specifier is D, the simplified bit value is interpreted as a decimal integer. The expanded bit value is a string of 0 and 1 digits that is the binary representation of the decimal integer. The number of characters in the expanded bit value is given by the expression $\lfloor \log_2 n \rfloor + 1$, where n is the value of the decimal integer.

The *length* of a bit string literal is the length of its string literal value. If a bit string literal includes the integer immediately preceding the base specifier, the length of the bit string literal is the value of the integer. Otherwise, the length is the number of characters in the expanded bit value.

The string literal value is obtained by adjusting the expanded bit value to the length of the bit string literal, as follows:

— If the length is equal to the number of characters in the expanded bit value, the string literal value is the expanded bit value itself.

— If the length is greater than the number of characters in the expanded bit value and the base specifier is B, UB, O, UO, X, UX, or D, the bit string value is obtained by concatenating a string of 0 digits to the left of the expanded bit value. The number of 0 digits in the string is such that the number of characters in the result of the concatenation is the length of the bit string literal.

— If the length is greater than the number of characters in the expanded bit value and the base specifier is SB, SO, or SX, the bit string value is obtained by concatenating to the left of the expanded bit value a string, each of whose characters is the leftmost character of the expanded bit value. The

number of characters in the string is such that the number of characters in the result of the concatenation is the length of the bit string literal.

— If the length is less than the number of characters in the expanded bit value and the base specifier is B, UB, O, UO, X, UX, or D, the bit string value is obtained by deleting sufficient characters from the left of the expanded bit value to yield a string whose length is the length of the bit string literal. It is an error if any of the characters so deleted is other than the digit 0.

— If the length is less than the number of characters in the expanded bit value and the base specifier is SB, SO, or SX, the bit string value is obtained by deleting sufficient characters from the left of the expanded bit value to yield a string whose length is the length of the bit string literal. It is an error if any of the characters so deleted differs from the leftmost remaining character.

Example:

```
B"1111_1111_1111"    --  Equivalent to the string literal "111111111111".
X"FFF"               --  Equivalent to B"1111_1111_1111".
O"777"               --  Equivalent to B"111_111_111".
X"777"               --  Equivalent to B"0111_0111_0111".

B"XXXX_01LH"         -- Equivalent to the string literal "XXXX01LH"
UO"27"               -- Equivalent to B"010_111"
UO"2C"               -- Equivalent to B"011_CCC"
SX"3W"               -- Equivalent to B"0011_WWWW"
D"35"                -- Equivalent to B"100011"

12UB"X1"             -- Equivalent to B"0000_0000_00X1"
12SB"X1"             -- Equivalent to B"XXXX_XXXX_XXX1"
12UX"F-"             -- Equivalent to B"0000_1111_----"
12SX"F-"             -- Equivalent to B"1111_1111_----"
12D"13"              -- Equivalent to B"0000_0000_1101"

12UX"000WWW"         -- Equivalent to B"WWWW_WWWW_WWWW"
12SX"FFFC00"         -- Equivalent to B"1100_0000_0000"
12SX"XXXX00"         -- Equivalent to B"XXXX_0000_0000"

8D"511"              -- Error
8UO"477"             -- Error
8SX"0FF"             -- Error
8SX"FXX"             -- Error

constant c1: STRING := B"1111_1111_1111";

constant c2: BIT_VECTOR := X"FFF";

type MVL is ('X', '0', '1', 'Z');
type MVL_VECTOR is array (NATURAL range <>) of MVL;
constant c3: MVL_VECTOR := O"777";

assert c1'LENGTH = 12 and c2'LENGTH = 12 and c3 = "111111111";
```

15.9 Comments

A comment is either a *single-line comment* or a *delimited comment*. A single-line comment starts with two adjacent hyphens and extends up to the end of the line. A delimited comment starts with a solidus (slash)

character immediately followed by an asterisk character and extends up to the first subsequent occurrence of an asterisk character immediately followed by a solidus character.

An occurrence of two adjacent hyphens within a delimited comment is not interpreted as the start of a single-line comment. Similarly, an occurrence of a solidus character immediately followed by an asterisk character within a single-line comment is not interpreted as the start of a delimited comment. Moreover, an occurrence of a solidus character immediately followed by an asterisk character within a delimited comment is not interpreted as the start of a nested delimited comment.

A single-line comment can appear on any line of a VHDL description and may contain any character except the format effectors vertical tab, carriage return, line feed, and form feed. A delimited comment can start on any line of a VHDL description and may finish on the same line or any subsequent line.

The presence or absence of comments has no influence on whether a description is legal or illegal. Furthermore, comments do not influence the execution of a simulation module; their sole purpose is to enlighten the human reader.

Examples:

```
--  The last sentence above echoes the Algol 68 report.

end;  --  Processing of LINE is complete.

-----------  The first two hyphens start the comment.

/* A long comment may be written
    on several consecutive lines */

x := 1;  /* Comments /* do not nest */
```

NOTE 1—Horizontal tabulation can be used in comments, after the starting characters, and is equivalent to one or more spaces (SPACE characters) (see 15.3).

NOTE 2—Comments may contain characters that, according to 15.2, are non-printing characters. Implementations may interpret the characters of a comment as members of ISO/IEC 8859-1:1998, or of any other character set; for example, an implementation may interpret multiple consecutive characters within a comment as single characters of a multi-byte character set.

15.10 Reserved words

The following identifiers are called *reserved words* and are reserved for significance in the language. For readability of this standard, the reserved words appear in lowercase boldface.

abs
access
after
alias
all
and
architecture
array
assert
assume
assume_guarantee
attribute

begin
block
body
buffer
bus

case
component
configuration
constant
context
cover

default
disconnect
downto

else
elsif
end
entity
exit

fairness
file
for
force
function

generate
generic
group
guarded

if
impure
in
inertial
inout
is

label
library
linkage
literal
loop

map
mod

nand
new
next
nor
not
null

of
on
open
or
others
out

package
parameter
port
postponed
procedure
process
property
protected
pure

range
record
register
reject
release
rem
report
restrict
restrict_guarantee
return
rol
ror

select
sequence
severity
signal
shared
sla
sll
sra
srl
strong
subtype

then
to
transport
type

unaffected
units
until
use

variable
vmode
vprop
vunit

wait
when
while
with

xnor
xor

A reserved word shall not be used as an explicitly declared identifier.

Within a PSL declaration, a PSL directive, or a PSL verification unit, PSL keywords are reserved words (see IEEE Std 1850-2005). A PSL keyword shall not be used as an identifier to declare a PSL declaration or a PSL verification unit. A PSL keyword that is a legal VHDL identifier may be used as an explicitly declared identifier other than to declare a PSL declaration or a PSL verification unit, but such a declaration is hidden within a PSL declaration, a PSL directive, or a PSL verification unit (see 12.3).

NOTE 1—Reserved words differing only in the use of corresponding uppercase and lowercase letters are considered as the same (see 15.4.2). The reserved words **range** and **subtype** are also used as the names of predefined attributes.

NOTE 2—An extended identifier whose sequence of characters inside the leading and trailing backslashes is identical to a reserved word is not a reserved word. For example, \next\ is a legal (extended) identifier and is not the reserved word **next**.

NOTE 3—The following reserved words are PSL keywords, that is, reserved identifiers in PSL:

assert	**default**	**restrict_guarantee**	**vprop**
assume	**fairness**	**sequence**	**vunit**
assume_guarantee	**property**	**strong**	
cover	**restrict**	**vmode**	

Their use in PSL is defined in IEEE Std 1850-2005. Other PSL keywords, reserved only within PSL declarations, PSL directives, and PSL verification units, are defined in IEEE Std 1850-2005.

15.11 Tool directives

A tool directive directs a tool to analyze, elaborate, execute, or otherwise process a description in a specified manner. A tool directive starts with a grave accent character and extends up to the end of the line.

tool_directive ::= ` identifier { graphic_character }

The identifier determines the form of processing to be performed by the tool. Apart from the standard tool directives (see Clause 24), the requirements, if any, on the location of a tool directive and on the graphic characters are implementation defined, as is the effect of the tool directive.

16. Predefined language environment

16.1 General

This clause describes the predefined attributes of VHDL and the packages that all VHDL implementations shall provide.

16.2 Predefined attributes

16.2.1 General

Predefined attributes denote values, functions, types, subtypes, signals, and ranges associated with various kinds of named entities. These attributes are described as follows. For each attribute, the following information is provided:

— The kind of attribute: value, type, subtype, range, function, or signal

— The prefixes for which the attribute is defined

— A description of the parameter or argument, if one exists

— The result of evaluating the attribute, and the result type (if applicable)

— Any further restrictions or comments that apply

For those predefined attributes that denote functions, the functions do not have named formal parameters; therefore, named association (see 6.5.7.1) cannot be used when invoking a function denoted by a predefined attribute.

16.2.2 Predefined attributes of types and objects

T'BASE	Kind:	Type.
	Prefix:	Any type or subtype T.
	Result:	The base type of T.
	Restrictions:	This attribute is allowed only as the prefix of the name of another attribute; for example, T'BASE'LEFT.
T'LEFT	Kind:	Value.
	Prefix:	Any scalar type or subtype T.
	Result type:	Same type as T.
	Result:	The left bound of T.
T'RIGHT	Kind:	Value.
	Prefix:	Any scalar type or subtype T.
	Result type:	Same type as T.
	Result:	The right bound of T.
T'HIGH	Kind:	Value.
	Prefix:	Any scalar type or subtype T.
	Result type:	Same type as T.
	Result:	The upper bound of T.

T'LOW	Kind:	Value.
	Prefix:	Any scalar type or subtype T.
	Result type:	Same type as T.
	Result:	The lower bound of T.
T'ASCENDING	Kind:	Value.
	Prefix:	Any scalar type or subtype T.
	Result type:	Type BOOLEAN
	Result:	It is TRUE if T is defined with an ascending range; FALSE otherwise.
T'IMAGE(X)	Kind:	Pure function.
	Prefix:	Any scalar type or subtype T.
	Parameter:	An expression whose type is the base type of T.
	Result type:	Type STRING.
	Result:	The string representation of the parameter value as defined in 5.7, but with the following differences. If T is an enumeration type or subtype and the parameter value is either an extended identifier or a character literal, the result is expressed with both a leading and trailing reverse solidus (backslash) (in the case of an extended identifier) or apostrophe (in the case of a character literal); in the case of an extended identifier that has a backslash, the backslash is doubled in the string representation. If T is an enumeration type or subtype and the parameter value is a basic identifier, then the result is expressed in lowercase characters. If T is a numeric type or subtype, the result is expressed as the decimal representation of the parameter value without underlines or leading or trailing zeros (except as necessary to form the image of a legal literal with the proper value); moreover, an exponent may (but is not required to) be present and the language does not define under what conditions it is or is not present. If the exponent is present, the “e” is expressed as a lowercase character. If T is a physical type or subtype, the result is expressed in terms of the primary unit of T unless the base type of T is TIME, in which case the result is expressed in terms of the resolution limit (see 5.2.4.2); in either case, if the unit is a basic identifier, the image of the unit is expressed in lowercase characters. If T is a floating-point type or subtype, the number of digits to the right of the decimal point corresponds to the standard form generated when the DIGITS parameter to TEXTIO.WRITE for type REAL is set to 0 (see 16.4).
	Restrictions:	It is an error if the parameter value does not belong to the subtype implied by the prefix.

T'VALUE(X)	Kind:	Pure function.
	Prefix:	Any scalar type or subtype T.
	Parameter:	An expression of type STRING.
	Result type:	The base type of T.
	Result:	The value of T whose string representation (as defined in 5.7) is given by the parameter. Leading and trailing whitespace is allowed and ignored. If T is a numeric type or subtype, the parameter shall be expressed either as a decimal literal or as a based literal, with the addition of an optional leading sign. If the sign is present, whitespace shall not occur between the sign and the remainder of the value. If T is a physical type or subtype, the parameter shall be expressed using a string representation of any of the unit names of T, with or without a leading abstract literal. The parameter shall have whitespace between any abstract literal and the unit name.
	Restrictions:	It is an error if the parameter is not a valid string representation of a literal of type T or if the result does not belong to the subtype implied by T.
T'POS(X)	Kind:	Pure function.
	Prefix:	Any discrete or physical type or subtype T.
	Parameter:	An expression whose type is the base type of T.
	Result type:	*universal_integer*.
	Result:	The position number of the value of the parameter.
	Restrictions:	It is an error if the value of the parameter does not belong to the subtype implied by the prefix.
T'VAL(X)	Kind:	Pure function.
	Prefix:	Any discrete or physical type or subtype T.
	Parameter:	An expression of any integer type.
	Result type:	The base type of T.
	Result:	The value whose position number is the *universal_integer* value corresponding to X.
	Restrictions:	It is an error if the result does not belong to the range T'LOW **to** T'HIGH.
T'SUCC(X)	Kind:	Pure function.
	Prefix:	Any discrete or physical type or subtype T.
	Parameter:	An expression whose type is the base type of T.
	Result type:	The base type of T.
	Result:	The value whose position number is one greater than that of the parameter.
	Restrictions:	An error occurs if X equals T'HIGH or if X does not belong to the range T'LOW **to** T'HIGH.

T'PRED(X)	Kind:	Pure function.
	Prefix:	Any discrete or physical type or subtype T.
	Parameter:	An expression whose type is the base type of T.
	Result type:	The base type of T.
	Result:	The value whose position number is one less than that of the parameter.
	Restrictions:	An error occurs if X equals T'LOW or if X does not belong to the range T'LOW **to** T'HIGH.
T'LEFTOF(X)	Kind:	Pure function.
	Prefix:	Any discrete or physical type or subtype T.
	Parameter:	An expression whose type is the base type of T.
	Result type:	The base type of T.
	Result:	The value that is to the left of the parameter in the range of T.
	Restrictions:	An error occurs if X equals T'LEFT or if X does not belong to the range T'LOW **to** T'HIGH.
T'RIGHTOF(X)	Kind:	Pure function.
	Prefix:	Any discrete or physical type or subtype T.
	Parameter:	An expression whose type is the base type of T.
	Result type:	The base type of T.
	Result:	The value that is to the right of the parameter in the range of T.
	Restrictions:	An error occurs if X equals T'RIGHT or if X does not belong to the range T'LOW **to** T'HIGH.
O'SUBTYPE	Kind:	Subtype.
	Prefix:	Any prefix O that is appropriate for an object, or an alias thereof.
	Result:	The fully constrained subtype that is the subtype of O, together with constraints defining any index ranges that are determined by application of the rules of 5.3.2.2. (If O is an alias for an object, then the result is determined by the declaration of O, not that of the object.)

NOTE 1—The relationship between the values of the LEFT, RIGHT, LOW, and HIGH attributes is expressed as follows:

		Ascending range	**Descending range**
T'LEFT	=	T'LOW	T'HIGH
T'RIGHT	=	T'HIGH	T'LOW

NOTE 2—For all values V of any scalar type T except a real type, the following relation holds:

```
V = T'Value(T'Image(V))
```

16.2.3 Predefined attributes of arrays

A'LEFT [(N)]	Kind:	Function.
	Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes an array subtype whose index ranges are defined by a constraint.
	Parameter:	A locally static expression of type *universal_integer*, the value of which shall not exceed the dimensionality of A. If omitted, it defaults to 1.
	Result type:	Type of the left bound of the Nth index range of A.
	Result:	Left bound of the Nth index range of A. (If A is an alias for an array object, then the result is the left bound of the Nth index range from the declaration of A, not that of the object.)
A'RIGHT [(N)]	Kind:	Function.
	Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes an array subtype whose index ranges are defined by a constraint.
	Parameter:	A locally static expression of type *universal_integer*, the value of which shall not exceed the dimensionality of A. If omitted, it defaults to 1.
	Result type:	Type of the Nth index range of A.
	Result:	Right bound of the Nth index range of A. (If A is an alias for an array object, then the result is the right bound of the Nth index range from the declaration of A, not that of the object.)
A'HIGH [(N)]	Kind:	Function.
	Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes an array subtype whose index ranges are defined by a constraint.
	Parameter:	A locally static expression of type *universal_integer*, the value of which shall not exceed the dimensionality of A. If omitted, it defaults to 1.
	Result type:	Type of the Nth index range of A.
	Result:	Upper bound of the Nth index range of A. (If A is an alias for an array object, then the result is the upper bound of the Nth index range from the declaration of A, not that of the object.)
A'LOW [(N)]	Kind:	Function.
	Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes an array subtype whose index ranges are defined by a constraint.
	Parameter:	A locally static expression of type *universal_integer*, the value of which shall not exceed the dimensionality of A. If omitted, it defaults to 1.
	Result type:	Type of the Nth index range of A.
	Result:	Lower bound of the Nth index range of A. (If A is an alias for an array object, then the result is the lower bound of the Nth index range from the declaration of A, not that of the object.)

A'RANGE [(N)]	Kind:	Range.
	Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes an array subtype whose index ranges are defined by a constraint.
	Parameter:	A locally static expression of type *universal_integer*, the value of which shall not exceed the dimensionality of A. If omitted, it defaults to 1.
	Result type:	The type of the Nth index range of A.
	Result:	The range A'LEFT(N) **to** A'RIGHT(N) if the Nth index range of A is ascending, or the range A'LEFT(N) **downto** A'RIGHT(N) if the Nth index range of A is descending. (If A is an alias for an array object, then the result is determined by the Nth index range from the declaration of A, not that of the object.)
A'REVERSE_RANGE [(N)]	Kind:	Range.
	Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes an array subtype whose index ranges are defined by a constraint.
	Parameter:	A locally static expression of type *universal_integer*, the value of which shall not exceed the dimensionality of A. If omitted, it defaults to 1.
	Result type:	The type of the Nth index range of A.
	Result:	The range A'RIGHT(N) **downto** A'LEFT(N) if the Nth index range of A is ascending, or the range A'RIGHT(N) **to** A'LEFT(N) if the Nth index range of A is descending. (If A is an alias for an array object, then the result is determined by the Nth index range from the declaration of A, not that of the object.)
A'LENGTH [(N)]	Kind:	Function.
	Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes an array subtype whose index ranges are defined by a constraint.
	Parameter:	A locally static expression of type *universal_integer*, the value of which shall not exceed the dimensionality of A. If omitted, it defaults to 1.
	Result type:	*universal_integer*.
	Result:	Number of values in the Nth index range; i.e., if the Nth index range of A is a null range, then the result is 0. Otherwise, the result is the value of T'POS(A'HIGH(N)) – T'POS(A'LOW(N)) + 1, where T is the subtype of the Nth index of A.
A'ASCENDING [(N)]	Kind:	Function.
	Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes an array subtype whose index ranges are defined by a constraint.
	Parameter:	A locally static expression of type *universal integer*, the value of which shall be greater than zero and shall not exceed the dimensionality of A. If omitted, it defaults to 1.
	Result type:	Type BOOLEAN.
	Result:	TRUE if the Nth index range of A is defined with an ascending range; FALSE otherwise.

A'ELEMENT	Kind:	Subtype.
	Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes an array subtype.
	Result:	If A is an array subtype, the result is the element subtype of A. If A is an array object, the result is the fully constrained element subtype that is the element subtype of A, together with constraints defining any index ranges that are determined by application of the rules of 5.3.2.2. (If A is an alias for an array object, then the result is determined by the declaration of A, not that of the object.)

16.2.4 Predefined attributes of signals

S'DELAYED [(T)]	Kind:	Signal.
	Prefix:	Any signal denoted by the static signal name S.
	Parameter:	A static expression of type TIME that evaluates to a nonnegative value. If omitted, it defaults to 0 ns.
	Result type:	The base type of S.
	Result:	A signal equivalent to signal S delayed T units of time.

Let R be of the same subtype as S, let T >= 0 ns, and let P be a process statement of the form

```
P: process (S)
   begin
      R <= transport S after T;
end process;
```

Assuming that the initial value of R is the same as the initial value of S, then the attribute 'DELAYED is defined such that S'DELAYED(T) = R for any T.

S'STABLE [(T)]	Kind:	Signal.
	Prefix:	Any signal denoted by the static signal name S.
	Parameter:	A static expression of type TIME that evaluates to a nonnegative value. If omitted, it defaults to 0 ns.
	Result type:	Type BOOLEAN.
	Result:	A signal that has the value TRUE when an event has not occurred on signal S for T units of time, and the value FALSE otherwise (see 14.7.3.4).
S'QUIET [(T)]	Kind:	Signal.
	Prefix:	Any signal denoted by the static signal name S.
	Parameter:	A static expression of type TIME that evaluates to a nonnegative value. If omitted, it defaults to 0 ns.
	Result type:	Type BOOLEAN.
	Result:	A signal that has the value TRUE when the signal has been quiet for T units of time, and the value FALSE otherwise (see 14.7.3.1).

S'TRANSACTION	Kind:	Signal.
	Prefix:	Any signal denoted by the static signal name S.
	Result type:	Type BIT.
	Result:	A signal whose value toggles to the inverse of its previous value in each simulation cycle in which signal S becomes active.
	Restrictions:	A description is erroneous if it depends on the initial value of S'TRANSACTION.
S'EVENT	Kind:	Function.
	Prefix:	Any signal denoted by the static signal name S.
	Result type:	Type BOOLEAN.
	Result:	A value that indicates whether an event has just occurred on signal S. Specifically:

For a scalar signal S, S'EVENT returns the value TRUE if an event has occurred on S during the current simulation cycle; otherwise, it returns the value FALSE.

For a composite signal S, S'EVENT returns TRUE if an event has occurred on any scalar subelement of S during the current simulation cycle; otherwise, it returns FALSE.

S'ACTIVE	Kind:	Function.
	Prefix:	Any signal denoted by the static signal name S.
	Result type:	Type BOOLEAN.
	Result:	A value that indicates whether signal S is active. Specifically:

For a scalar signal S, S'ACTIVE returns the value TRUE if signal S is active during the current simulation cycle; otherwise, it returns the value FALSE.

For a composite signal S, S'ACTIVE returns TRUE if any scalar subelement of S is active during the current simulation cycle; otherwise, it returns FALSE.

S'LAST_EVENT	Kind:	Function.
	Prefix:	Any signal denoted by the static signal name S.
	Result type:	Type TIME.
	Result:	The amount of time that has elapsed since the last event occurred on signal S. Specifically:

For a signal S, S'LAST_EVENT returns the smallest value T of type TIME such that S'EVENT = TRUE during any simulation cycle at time NOW – T, if such a value exists; otherwise, it returns TIME'HIGH.

S'LAST_ACTIVE	Kind:	Function.
	Prefix:	Any signal denoted by the static signal name S.
	Result type:	Type TIME.
	Result:	The amount of time that has elapsed since the last time at which signal S was active. Specifically:

For a signal S, S'LAST_ACTIVE returns the smallest value T of type TIME such that S'ACTIVE = TRUE during any simulation cycle at time NOW – T, if such value exists; otherwise, it returns TIME'HIGH.

S'LAST_VALUE	Kind:	Function.
	Prefix:	Any signal denoted by the static signal name S.
	Result type:	The base type of S.
	Result:	For a signal S, if an event has occurred on S in any simulation cycle, S'LAST_VALUE returns the value of S prior to the update of S in the last simulation cycle in which an event occurred; otherwise, S'LAST_VALUE returns the current value of S.
S'DRIVING	Kind:	Function.
	Prefix:	Any signal denoted by the static signal name S.
	Result type:	Type BOOLEAN.
	Result:	If the prefix denotes a scalar signal, the result is FALSE if the current value of the driver for S in the current process is determined by the null transaction; TRUE otherwise. If the prefix denotes a composite signal, the result is TRUE if and only if R'DRIVING is TRUE for every scalar subelement R of S; FALSE otherwise. If the prefix denotes a null slice of a signal, the result is TRUE.
	Restrictions:	This attribute is available only from within a process, a concurrent statement with an equivalent process, or a subprogram. If the prefix denotes a port, it is an error if the port does not have a mode of **inout**, **out**, or **buffer**. It is also an error if the attribute name appears in a subprogram body that is not a declarative item contained within a process statement and the prefix is not a formal parameter of the given subprogram or of a parent of that subprogram. Finally, it is an error if the prefix denotes a subprogram formal parameter whose mode is not **inout** or **out**.
S'DRIVING_VALUE	Kind:	Function.
	Prefix:	Any signal denoted by the static signal name S.
	Result type:	The base type of S.
	Result:	If S is a scalar signal, the result is the current value of the driver for S in the current process. If S is a composite signal, the result is the aggregate of the values of R'DRIVING_VALUE for each element R of S. If S is a null slice, the result is a null slice.
	Restrictions:	This attribute is available only from within a process, a concurrent statement with an equivalent process, or a subprogram. If the prefix denotes a port, it is an error if the port does not have a mode of **inout**, **out**, or **buffer**. It is also an error if the attribute name appears in a subprogram body that is not a declarative item contained within a process statement and the prefix is not a formal parameter of the given subprogram or of a parent of that subprogram. Finally, it is an error if the prefix denotes a subprogram formal parameter whose mode is not **inout** or **out**, or if S'DRIVING is FALSE at the time of the evaluation of S'DRIVING_VALUE.

NOTE 1—Since the attributes S'EVENT, S'ACTIVE, S'LAST_EVENT, S'LAST_ACTIVE, and S'LAST_VALUE are functions, not signals, they cannot cause the execution of a process, even though the value returned by such a function may change dynamically. It is thus recommended that the equivalent signal-valued attributes S'STABLE and S'QUIET, or expressions involving those attributes, be used in concurrent contexts such as guard conditions or concurrent signal assignments. Similarly, function STANDARD.NOW should not be used in concurrent contexts.

NOTE 2—S'DELAYED(0 ns) is not equal to S during any simulation cycle where S'EVENT is true.

NOTE 3—S'STABLE(0 ns) = (S'DELAYED(0 ns) = S), and S'STABLE(0 ns) is FALSE only during a simulation cycle in which S has had a transaction.

NOTE 4—For a given simulation cycle, S'QUIET(0 ns) is TRUE if and only if S is quiet for that simulation cycle.

NOTE 5—If S'STABLE(T) is FALSE, then, by definition, for some t where 0 ns < t < T, S'DELAYED(t) /= S.

NOTE 6—If T_s is the smallest value such that S'STABLE (T_s) is FALSE, then for all t where 0 ns < t < T_s, S'DELAYED(t) = S.

NOTE 7—S'EVENT should not be used within a postponed process (or a concurrent statement that has an equivalent postponed process) to determine if the prefix signal S caused the process to resume. However, S'LAST_EVENT = 0 ns can be used for this purpose.

NOTE 8—For a composite signal S, if an event on S as a whole is caused by an event on a subelement of S, the value of S'LAST_VALUE is the whole value of S before the update of the subelement. That value includes subelement values that may not have changed.

16.2.5 Predefined attributes of named entities

E'SIMPLE_NAME	Kind:	Value.
	Prefix:	Any named entity as defined in 7.2.
	Result type:	Type STRING.
	Result:	The simple name, character literal, or operator symbol of the named entity, without leading or trailing whitespace or quotation marks but with apostrophes (in the case of a character literal) and both a leading and trailing reverse solidus (backslash) (in the case of an extended identifier). In the case of a simple name or operator symbol, the characters are converted to their lowercase equivalents. In the case of an extended identifier, the case of the identifier is preserved, and any reverse solidus characters appearing as part of the identifier are represented with two consecutive reverse solidus characters.

E'INSTANCE_NAME

Kind: Value.

Prefix: Any named entity other than the local ports and generics of a component declaration.

Result type: Type STRING.

Result: A string describing the hierarchical path starting at the root of the design hierarchy and descending to the named entity, including the names of instantiated design entities. Specifically:

The result string has the following syntax:

```
instance_name ::=  package_based_path | full_instance_based_path

package_based_path ::=
    leader library_logical_name leader
        { package_path_instance_element leader }
        [ local_item_name ]

package_path_instance_element ::=
      subprogram_designator signature
    | variable_simple_name
    | package_simple_name

full_instance_based_path ::=  leader full_path_to_instance [ local_item_name ]

full_path_to_instance ::=  { full_path_instance_element leader }

local_item_name ::=
      simple_name
    | character_literal
    | operator_symbol

full_path_instance_element ::=
    [ component_instantiation_label @ ]
        entity_simple_name ( architecture_simple_name )
    | block_label
    | generate_label
    | process_label
    | loop_label
    | subprogram_designator signature
    | variable_simple_name
    | package_simple_name

generate_label ::=  generate_label [ ( literal ) ]

process_label ::=  [ process_label ]

loop_label ::=  [ loop_label ]

leader ::=  :
```

Package-based paths identify items declared within package library units. Full-instance-based paths identify items within an elaborated design hierarchy.

A library logical name denotes a library (see 13.2). Since it is possible for multiple logical names to denote the same library, it is possible that the library logical name not be unique.

The local item name in E'INSTANCE_NAME equals E'SIMPLE_NAME, unless E denotes a library, package, subprogram, label, or variable of a protected type. In this latter case, the package-based path or full-instance-based path, as appropriate, will not contain a local item name.

There is one package path instance element for each subprogram body, shared variable of a protected type, or nested package in the package library unit between the package declaration or package body of the package library unit and the named entity denoted by the prefix. Similarly, there is one full path instance element for each component instantiation, block statement, generate statement, process statement, loop statement, subprogram body, variable of a protected type, or package in the design hierarchy between the root design entity and the named entity denoted by the prefix.

In a full path instance element, the architecture simple name shall denote an architecture associated with the entity declaration designated by the entity simple name; furthermore, the component instantiation label (and the commercial at character following it) are required unless the entity simple name and the architecture simple name together denote the root design entity.

The literal in a generate label is required if the label denotes a for generate statement; the literal shall denote one of the values of the generate parameter.

A process statement with no label is denoted by an empty process label. Similarly, a loop statement with no label is denoted by an empty loop label.

The signature occurring after a subprogram designator in the result of the 'INSTANCE_NAME or 'PATH_NAME attribute shall match the parameter and result type profile of the subprogram. Each type mark in the signature is the type mark of the subtype indication of the corresponding formal parameter, or the return type mark, as appropriate, in the subprogram declaration.

All characters in basic identifiers appearing in the result are converted to their lowercase equivalents. Both a leading and trailing reverse solidus surround an extended identifier appearing in the result; any reverse solidus characters appearing as part of the identifier are represented with two consecutive reverse solidus characters.

E'PATH_NAME

Kind: Value.

Prefix: Any named entity other than the local ports and generics of a component declaration.

Result type: Type STRING.

Result: A string describing the hierarchical path starting at the root of the design hierarchy and descending to the named entity, excluding the name of instantiated design entities. Specifically:

The result string has the following syntax:

path_name ::= package_based_path | instance_based_path

instance_based_path ::=
leader path_to_instance [local_item_name]

path_to_instance ::= { path_instance_element leader }

path_instance_element ::=
*component_instantiation*_label
| *entity*_simple_name
| *block*_label
| generate_label
| process_label
| loop_label
| *subprogram*_designator signature
| *variable*_simple_name
| *package*_simple_name

Package-based paths identify items declared within package library units. Instance-based paths identify items within an elaborated design hierarchy.

The local item name in E'PATH_NAME equals E'SIMPLE_NAME, unless E denotes a library, package, subprogram, label, or variable of a protected type. In this latter case, the package-based path or instance-based path, as appropriate, will not contain a local item name.

There is one package path instance element for each subprogram body or shared variable of a protected type or nested package in the package library unit between the package declaration or package body of the package library unit and the named entity denoted by the prefix. Similarly, there is one path instance element for each component instantiation, block statement, generate statement, process statement, loop statement, subprogram body, variable of a protected type, or package in the design hierarchy between the root design entity and the named entity denoted by the prefix.

Examples:

```
library Lib;   --  All design units are in this library:
package P is   --  P'PATH_NAME = ":lib:p:"
               --  P'INSTANCE_NAME = ":lib:p:"
   procedure Proc (F: inout INTEGER);
      --  Proc'PATH_NAME = ":lib:p:proc [integer]:"
      --  Proc'INSTANCE_NAME = ":lib:p:proc [integer]:"
   constant C: INTEGER := 42;   --  C'PATH_NAME = ":lib:p:c"
end package P;                  --  C'INSTANCE_NAME = ":lib:p:c"

package body P is
   procedure Proc (F: inout INTEGER) is
      variable x: INTEGER;   --  x'PATH_NAME = ":lib:p:proc [integer]:x"
   begin                     --  x'INSTANCE_NAME = ":lib:p:proc [integer]:x"
```

```
      .
      .
      .
   end;
end;

library Lib;
use Lib.P.all;              --  Assume that E is in Lib and
entity E is                 --  E is the top-level design entity:
                            --  E'PATH_NAME = ":e:"
                            --  E'INSTANCE_NAME = ":e(a):"
   generic (G: INTEGER);    --  G'PATH_NAME = ":e:g"
                            --  G'INSTANCE_NAME = ":e(a):g"
   port (P: in INTEGER);    --  P'PATH_NAME = ":e:p"
end entity E;               --  P'INSTANCE_NAME = ":e(a):p"

architecture A of E is
   signal S: BIT_VECTOR (1 to G);   --  S'PATH_NAME = ":e:s"
                                    --  S'INSTANCE_NAME = ":e(a):s"
   procedure Proc1 (signal sp1: NATURAL; C: out INTEGER) is
      --  Proc1'PATH_NAME = ":e:proc1[natural,integer]:"
      --  Proc1'INSTANCE_NAME = ":e(a):proc1[natural,integer]:"
      --  C'PATH_NAME = ":e:proc1[natural,integer]:c"
      --  C'INSTANCE_NAME = ":e(a):proc1[natural,integer]:c"
      variable max: DELAY_LENGTH;
         --  max'PATH_NAME = ":e:proc1[natural,integer]:max"
         --  max'INSTANCE_NAME = ":e(a):proc1[natural,integer]:max"
   begin
      max := sp1 * ns;
      wait on sp1 for max;
      c := sp1;
   end procedure Proc1;

begin
   p1: process
      variable T: INTEGER := 12;    --  T'PATH_NAME =  :e:p1:t"
   begin                            --  T'INSTANCE_NAME = ":e(a):p1:t"
      .
      .
      .
   end process p1;

   process
      variable T: INTEGER := 12;    --  T'PATH_NAME = ":e::t"
   begin                            --  T'INSTANCE_NAME = ":e(a)::t"
      .
      .
      .
   end process ;
end architecture;

entity Bottom is
   generic (GBottom: INTEGER);
   port (PBottom: INTEGER);
end entity Bottom;

architecture BottomArch of Bottom is
   signal SBottom: INTEGER;
begin
```



```
   ProcessBottom: process
      variable V: INTEGER;
   begin
      if GBottom = 4 then
         assert V'Simple_Name = "v"
                and V'Path_Name = ":top:b1:b2:g1(4):b3:l1:processbottom:v"
                and V'Instance_Name =
      ":top(top):b1:b2:g1(4):b3:l1@bottom(bottomarch):processbottom:v";
         assert GBottom'Simple_Name = "gbottom"
                and GBottom'Path_Name = ":top:b1:b2:g1(4):b3:l1:gbottom"
                and GBottom'Instance_Name =
                ":top(top):b1:b2:g1(4):b3:l1@bottom(bottomarch):gbottom";

      elsif GBottom = -1 then
         assert V'Simple_Name = "v"
                and V'Path_Name = ":top:l2:processbottom:v"
                and V'Instance_Name =
                ":top(top):l2@bottom(bottomarch):processbottom:v";
         assert GBottom'Simple_Name = "gbottom"
                and GBottom'Path_Name = ":top:l2:gbottom"
                and GBottom'Instance_Name =
                ":top(top):l2@bottom(bottomarch):gbottom";
      end if;
      wait;
   end process ProcessBottom;
end architecture BottomArch;

entity Top is end Top;

architecture Top of Top is
   component BComp is
      generic (GComp: INTEGER);
      port (PComp: INTEGER);

   end component BComp;
   signal S: INTEGER;
begin
   B1: block
      signal S: INTEGER;
   begin
      B2: block
         signal S: INTEGER;
      begin
         G1: for I in 1 to 10 generate
            B3: block
               signal S: INTEGER;
               for L1: BComp use entity Work.Bottom(BottomArch)
                  generic map (GBottom => GComp)
                  port map (PBottom => PComp);
            begin
               L1: BComp generic map (I) port map (S);
               P1: process
                  variable V: INTEGER;
               begin
                  if I = 7 then
                     assert V'Simple_Name = "v"
                            and V'Path_Name = ":top:b1:b2:g1(7):b3:p1:v"
                            and V'Instance_Name =
                                ":top(top):b1:b2:g1(7):b3:p1:v";
```

```
                    assert P1'Simple_Name = "p1"
                           and P1'Path_Name = ":top:b1:b2:g1(7):b3:p1:"
                           and P1'Instance_Name =
                               ":top(top):b1:b2:g1(7):b3:p1:";
                    assert S'Simple_Name = "s"
                           and S'Path_Name = ":top:b1:b2:g1(7):b3:s"
                           and S'Instance_Name =
                               ":top(top):b1:b2:g1(7):b3:s";
                    assert B1.S'Simple_Name = "s"
                           and B1.S'Path_Name = ":top:b1:s"
                           and B1.S'Instance_Name = ":top(top):b1:s";
                end if;
                wait;
             end process P1;
          end block B3;
       end generate;
    end block B2;
 end block B1;
 L2: BComp generic map (-1) port map (S);
end architecture Top;

configuration TopConf of Top is
 for Top
    for L2: BComp use
       entity Work.Bottom(BottomArch)
          generic map (GBottom => GComp)
          port map (PBottom => PComp);
    end for;
 end for;
end configuration TopConf;
```

NOTE 1—The values of E'PATH_NAME and E'INSTANCE_NAME are not unique. Specifically, named entities in two different, unlabeled processes may have the same path names or instance names. Overloaded subprograms, and named entities within them, may also have the same path names or instance names.

NOTE 2—If the prefix to the attributes 'SIMPLE_NAME, 'PATH_NAME, or 'INSTANCE_NAME denotes an alias, the result is respectively the simple name, path name or instance name of the alias. See 8.6.

16.3 Package STANDARD

Package STANDARD predefines a number of types, subtypes, and functions. An implicit context clause naming this package is assumed to exist at the beginning of each design unit. Package STANDARD must not be modified by the user.

The operations that are predefined for the types declared for package STANDARD are given in comments since they are implicitly declared. Italics are used for pseudo-names of anonymous types (such as *universal_integer*), formal parameters, and undefined information (such as *implementation_defined*).

```
package STANDARD is

   --  Predefined enumeration types:
   type BOOLEAN is (FALSE, TRUE);

   --  The predefined operations for this type are as follows:

   --  function "and"  (anonymous, anonymous: BOOLEAN) return BOOLEAN;
   --  function "or"   (anonymous, anonymous: BOOLEAN) return BOOLEAN;
```



```
--  function "nand" (anonymous, anonymous: BOOLEAN) return BOOLEAN;
--  function "nor"  (anonymous, anonymous: BOOLEAN) return BOOLEAN;
--  function "xor"  (anonymous, anonymous: BOOLEAN) return BOOLEAN;
--  function "xnor" (anonymous, anonymous: BOOLEAN) return BOOLEAN;

--  function "not"  (anonymous: BOOLEAN) return BOOLEAN;

--  function "="    (anonymous, anonymous: BOOLEAN) return BOOLEAN;
--  function "/="   (anonymous, anonymous: BOOLEAN) return BOOLEAN;
--  function "<"    (anonymous, anonymous: BOOLEAN) return BOOLEAN;
--  function "<="   (anonymous, anonymous: BOOLEAN) return BOOLEAN;
--  function ">"    (anonymous, anonymous: BOOLEAN) return BOOLEAN;
--  function ">="   (anonymous, anonymous: BOOLEAN) return BOOLEAN;

--  function MINIMUM (L, R: BOOLEAN) return BOOLEAN;
--  function MAXIMUM (L, R: BOOLEAN) return BOOLEAN;

--  function RISING_EDGE  (signal S: BOOLEAN) return BOOLEAN;
--  function FALLING_EDGE (signal S: BOOLEAN) return BOOLEAN;

type BIT is ('0', '1');

--  The predefined operations for this type are as follows:

--  function "and"  (anonymous, anonymous: BIT) return BIT;
--  function "or"   (anonymous, anonymous: BIT) return BIT;
--  function "nand" (anonymous, anonymous: BIT) return BIT;
--  function "nor"  (anonymous, anonymous: BIT) return BIT;
--  function "xor"  (anonymous, anonymous: BIT) return BIT;
--  function "xnor" (anonymous, anonymous: BIT) return BIT;

--  function "not"  (anonymous: BIT) return BIT;

--  function "="    (anonymous, anonymous: BIT) return BOOLEAN;
--  function "/="   (anonymous, anonymous: BIT) return BOOLEAN;
--  function "<"    (anonymous, anonymous: BIT) return BOOLEAN;
--  function "<="   (anonymous, anonymous: BIT) return BOOLEAN;
--  function ">"    (anonymous, anonymous: BIT) return BOOLEAN;
--  function ">="   (anonymous, anonymous: BIT) return BOOLEAN;

--  function "?="   (anonymous, anonymous: BIT) return BIT;
--  function "?/="  (anonymous, anonymous: BIT) return BIT;
--  function "?<"   (anonymous, anonymous: BIT) return BIT;
--  function "?<="  (anonymous, anonymous: BIT) return BIT;
--  function "?>"   (anonymous, anonymous: BIT) return BIT;
--  function "?>="  (anonymous, anonymous: BIT) return BIT;

--  function MINIMUM (L, R: BIT) return BIT;
--  function MAXIMUM (L, R: BIT) return BIT;

--  function "??"   (anonymous: BIT) return BOOLEAN;

--  function RISING_EDGE  (signal S: BIT) return BOOLEAN;
--  function FALLING_EDGE (signal S: BIT) return BOOLEAN;
```

```
type CHARACTER is (

   NUL,    SOH,    STX,    ETX,    EOT,    ENQ,    ACK,    BEL,
   BS,     HT,     LF,     VT,     FF,     CR,     SO,     SI,
   DLE,    DC1,    DC2,    DC3,    DC4,    NAK,    SYN,    ETB,
   CAN,    EM,     SUB,    ESC,    FSP,    GSP,    RSP,    USP,

   ' ',    '!',    '"',    '#',    '$',    '%',    '&',    ''',
   '(',    ')',    '*',    '+',    ',',    '-',    '.',    '/',
   '0',    '1',    '2',    '3',    '4',    '5',    '6',    '7',
   '8',    '9',    ':',    ';',    '<',    '=',    '>',    '?',

   '@',    'A',    'B',    'C',    'D',    'E',    'F',    'G',
   'H',    'I',    'J',    'K',    'L',    'M',    'N',    'O',
   'P',    'Q',    'R',    'S',    'T',    'U',    'V',    'W',
   'X',    'Y',    'Z',    '[',    '\',    ']',    '^',    '_',

   '`',    'a',    'b',    'c',    'd',    'e',    'f',    'g',
   'h',    'i',    'j',    'k',    'l',    'm',    'n',    'o',
   'p',    'q',    'r',    's',    't',    'u',    'v',    'w',
   'x',    'y',    'z',    '{',    '|',    '}',    '~',    DEL,

   C128,   C129,   C130,   C131,   C132,   C133,   C134,   C135,
   C136,   C137,   C138,   C139,   C140,   C141,   C142,   C143,
   C144,   C145,   C146,   C147,   C148,   C149,   C150,   C151,
   C152,   C153,   C154,   C155,   C156,   C157,   C158,   C159,

   ' ',[9]  '¡',    '¢',    '£',    '¤',    '¥',    '¦',    '§',
   '¨',    '©',    'ª',    '«',    '¬',    '-',[10]  '®',    '¯',
   '°',    '±',    '²',    '³',    '´',    'µ',    '¶',    '·',
   '¸',    '¹',    'º',    '»',    '¼',    '½',    '¾',    '¿',

   'À',    'Á',    'Â',    'Ã',    'Ä',    'Å',    'Æ',    'Ç',
   'È',    'É',    'Ê',    'Ë',    'Ì',    'Í',    'Î',    'Ï',
   'Ð',    'Ñ',    'Ò',    'Ó',    'Ô',    'Õ',    'Ö',    '×',
   'Ø',    'Ù',    'Ú',    'Û',    'Ü',    'Ý',    'Þ',    'ß',

   'à',    'á',    'â',    'ã',    'ä',    'å',    'æ',    'ç',
   'è',    'é',    'ê',    'ë',    'ì',    'í',    'î',    'ï',
   'ð',    'ñ',    'ò',    'ó',    'ô',    'õ',    'ö',    '÷',
   'ø',    'ù',    'ú',    'û',    'ü',    'ý',    'þ',    'ÿ');

--  The predefined operations for this type are as follows:

--  function "="  (anonymous, anonymous: CHARACTER)
--                                       return BOOLEAN;
--  function "/=" (anonymous, anonymous: CHARACTER)
--                                       return BOOLEAN;
--  function "<"  (anonymous, anonymous: CHARACTER)
--                                       return BOOLEAN;
```

[9]The nonbreaking space character.
[10]The soft hyphen character.


```
--  function "<=" (anonymous, anonymous: CHARACTER)
--                                       return BOOLEAN;
--  function ">"  (anonymous, anonymous: CHARACTER)
--                                       return BOOLEAN;
--  function ">=" (anonymous, anonymous: CHARACTER)
--                                       return BOOLEAN;

--  function MINIMUM (L, R: CHARACTER) return CHARACTER;
--  function MAXIMUM (L, R: CHARACTER) return CHARACTER;

type SEVERITY_LEVEL is (NOTE, WARNING, ERROR, FAILURE);

--  The predefined operations for this type are as follows:

--  function "="  (anonymous, anonymous: SEVERITY_LEVEL)
--                                       return BOOLEAN;
--  function "/=" (anonymous, anonymous: SEVERITY_LEVEL)
--                                       return BOOLEAN;
--  function "<"  (anonymous, anonymous: SEVERITY_LEVEL)
--                                       return BOOLEAN;
--  function "<=" (anonymous, anonymous: SEVERITY_LEVEL)
--                                       return BOOLEAN;
--  function ">"  (anonymous, anonymous: SEVERITY_LEVEL)
--                                       return BOOLEAN;
--  function ">=" (anonymous, anonymous: SEVERITY_LEVEL)
--                                       return BOOLEAN;

--  function MINIMUM (L, R: SEVERITY_LEVEL) return SEVERITY_LEVEL;
--  function MAXIMUM (L, R: SEVERITY_LEVEL) return SEVERITY_LEVEL;

--  type universal_integer  is range implementation_defined;

--  The predefined operations for this type are as follows:

--  function "="   (anonymous, anonymous: universal_integer)
--                                       return BOOLEAN;
--  function "/="  (anonymous, anonymous: universal_integer)
--                                       return BOOLEAN;
--  function "<"   (anonymous, anonymous: universal_integer)
--                                       return BOOLEAN;
--  function "<="  (anonymous, anonymous: universal_integer)
--                                       return BOOLEAN;
--  function ">"   (anonymous, anonymous: universal_integer)
--                                       return BOOLEAN;
--  function ">="  (anonymous, anonymous: universal_integer)
--                                       return BOOLEAN;
--  function "+"   (anonymous: universal_integer)
--                                       return universal_integer;
--  function "-"   (anonymous: universal_integer)
--                                       return universal_integer;
--  function "abs" (anonymous: universal_integer)
--                                       return universal_integer;

--  function "+"   (anonymous, anonymous: universal_integer)
```

```
--                                          return universal_integer;
--  function "-"   (anonymous, anonymous: universal_integer)
--                                          return universal_integer;
--  function "*"   (anonymous, anonymous: universal_integer)
--                                          return universal_integer;
--  function "/"   (anonymous, anonymous: universal_integer)
--                                          return universal_integer;
--  function "mod" (anonymous, anonymous: universal_integer)
--                                          return universal_integer;
--  function "rem" (anonymous, anonymous: universal_integer)
--                                          return universal_integer;

--  function MINIMUM (L, R: universal_integer)
--                                          return universal_integer;
--  function MAXIMUM (L, R: universal_integer)
--                                          return universal_integer;

--  type universal_real  is range implementation_defined;

--  The predefined operations for this type are as follows:

--  function "="   (anonymous, anonymous: universal_real)
--                                           return BOOLEAN;
--  function "/="  (anonymous, anonymous: universal_real)
--                                           return BOOLEAN;
--  function "<"   (anonymous, anonymous: universal_real)
--                                           return BOOLEAN;
--  function "<="  (anonymous, anonymous: universal_real)
--                                           return BOOLEAN;
--  function ">"   (anonymous, anonymous: universal_real)
--                                           return BOOLEAN;
--  function ">="  (anonymous, anonymous: universal_real)
--                                           return BOOLEAN;

--  function "+"   (anonymous: universal_real)
--                                           return universal_real;
--  function "-"   (anonymous: universal_real)
--                                           return universal_real;
--  function "abs" (anonymous: universal_real)
--                                           return universal_real;

--  function "+"   (anonymous, anonymous: universal_real)
--                                           return universal_real;
--  function "-"   (anonymous, anonymous: universal_real)
--                                           return universal_real;
--  function "*"   (anonymous, anonymous: universal_real)
--                                           return universal_real;
--  function "/"   (anonymous, anonymous: universal_real)
--                                           return universal_real;

--  function "*"   (anonymous: universal_real;
--                  anonymous: universal_integer)
--                                           return universal_real;
--  function "*"   (anonymous: universal_integer;
```

```
--                  anonymous: universal_real)
--                                      return universal_real;
--  function "/"   (anonymous: universal_real;
--                  anonymous: universal_integer)
--                                      return universal_real;

--  function MINIMUM (L, R: universal_real) return universal_real;
--  function MAXIMUM (L, R: universal_real) return universal_real;

--  Predefined numeric types:
type INTEGER is range implementation_defined;

--  The predefined operations for this type are as follows:

--  function "**"  (anonymous: universal_integer;
--                  anonymous: INTEGER) return universal_integer;
--  function "**"  (anonymous: universal_real;
--                  anonymous: INTEGER) return universal_real;

--  function "="   (anonymous, anonymous: INTEGER) return BOOLEAN;
--  function "/="  (anonymous, anonymous: INTEGER) return BOOLEAN;
--  function "<"   (anonymous, anonymous: INTEGER) return BOOLEAN;
--  function "<="  (anonymous, anonymous: INTEGER) return BOOLEAN;
--  function ">"   (anonymous, anonymous: INTEGER) return BOOLEAN;
--  function ">="  (anonymous, anonymous: INTEGER) return BOOLEAN;
--  function "+"   (anonymous: INTEGER) return INTEGER;
--  function "-"   (anonymous: INTEGER) return INTEGER;
--  function "abs" (anonymous: INTEGER) return INTEGER;

--  function "+"   (anonymous, anonymous: INTEGER) return INTEGER;
--  function "-"   (anonymous, anonymous: INTEGER) return INTEGER;
--  function "*"   (anonymous, anonymous: INTEGER) return INTEGER;
--  function "/"   (anonymous, anonymous: INTEGER) return INTEGER;
--  function "mod" (anonymous, anonymous: INTEGER) return INTEGER;
--  function "rem" (anonymous, anonymous: INTEGER) return INTEGER;

--  function "**"  (anonymous: INTEGER; anonymous: INTEGER)
--                                      return INTEGER;

--  function MINIMUM (L, R: INTEGER) return INTEGER;
--  function MAXIMUM (L, R: INTEGER) return INTEGER;

type REAL is range implementation_defined;

--  The predefined operations for this type are as follows:

--  function "="   (anonymous, anonymous: REAL) return BOOLEAN;
--  function "/="  (anonymous, anonymous: REAL) return BOOLEAN;
--  function "<"   (anonymous, anonymous: REAL) return BOOLEAN;
--  function "<="  (anonymous, anonymous: REAL) return BOOLEAN;
--  function ">"   (anonymous, anonymous: REAL) return BOOLEAN;
--  function ">="  (anonymous, anonymous: REAL) return BOOLEAN;

--  function "+"   (anonymous: REAL) return REAL;
```

```
--  function "-"   (anonymous: REAL) return REAL;
--  function "abs" (anonymous: REAL) return REAL;

--  function "+"   (anonymous, anonymous: REAL) return REAL;
--  function "-"   (anonymous, anonymous: REAL) return REAL;
--  function "*"   (anonymous, anonymous: REAL) return REAL;
--  function "/"   (anonymous, anonymous: REAL) return REAL;

--  function "**"  (anonymous: REAL; anonymous: INTEGER) return REAL;

--  function MINIMUM (L, R: REAL) return REAL;
--  function MAXIMUM (L, R: REAL) return REAL;

--  Predefined type TIME:

type TIME is range implementation_defined
   units
      fs;                       --  femtosecond
      ps   =  1000 fs;          --  picosecond
      ns   =  1000 ps;          --  nanosecond
      us   =  1000 ns;          --  microsecond
      ms   =  1000 us;          --  millisecond
      sec  =  1000 ms;          --  second
      min  =  60 sec;           --  minute
      hr   =  60 min;           --  hour
   end units;

--  The predefined operations for this type are as follows:

--  function "="   (anonymous, anonymous: TIME) return BOOLEAN;
--  function "/="  (anonymous, anonymous: TIME) return BOOLEAN;
--  function "<"   (anonymous, anonymous: TIME) return BOOLEAN;
--  function "<="  (anonymous, anonymous: TIME) return BOOLEAN;
--  function ">"   (anonymous, anonymous: TIME) return BOOLEAN;
--  function ">="  (anonymous, anonymous: TIME) return BOOLEAN;
--  function "+"   (anonymous: TIME) return TIME;
--  function "-    (anonymous: TIME) return TIME;
--  function "abs" (anonymous: TIME) return TIME;

--  function "+"   (anonymous, anonymous: TIME) return TIME;
--  function "-"   (anonymous, anonymous: TIME) return TIME;

--  function "*"   (anonymous: TIME;    anonymous: INTEGER)
--                                        return TIME;
--  function "*"   (anonymous: TIME;    anonymous: REAL)
--                                        return TIME;
--  function "*"   (anonymous: INTEGER; anonymous: TIME)
--                                        return TIME;
--  function "*"   (anonymous: REAL;    anonymous: TIME)
--                                        return TIME;
--  function "/"   (anonymous: TIME;    anonymous: INTEGER)
--                                        return TIME;
--  function "/"   (anonymous: TIME;    anonymous: REAL)
--                                        return TIME;
```

```
--  function "/"   (anonymous, anonymous: TIME)
--                                        return universal_integer;

--  function "mod" (anonymous, anonymous: TIME) return TIME;
--  function "rem" (anonymous, anonymous: TIME) return TIME;

--  function MINIMUM (L, R: TIME) return TIME;
--  function MAXIMUM (L, R: TIME) return TIME;

subtype DELAY_LENGTH is TIME range 0 fs to TIME'HIGH;

--  A function that returns the current simulation time, T_c,
-- (see 14.7.5.1):

impure function NOW return DELAY_LENGTH;

--  Predefined numeric subtypes:

subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;

--  Predefined array types:

type STRING is array (POSITIVE range <>) of CHARACTER;

--  The predefined operations for these types are as follows:

--  function "="  (anonymous, anonymous: STRING) return BOOLEAN;
--  function "/=" (anonymous, anonymous: STRING) return BOOLEAN;
--  function "<"  (anonymous, anonymous: STRING) return BOOLEAN;
--  function "<=" (anonymous, anonymous: STRING) return BOOLEAN;
--  function ">"  (anonymous, anonymous: STRING) return BOOLEAN;
--  function ">=" (anonymous, anonymous: STRING) return BOOLEAN;

--  function "&"  (anonymous: STRING;    anonymous: STRING)
--                                       return STRING;
--  function "&"  (anonymous: STRING;    anonymous: CHARACTER)
--                                       return STRING;
--  function "&"  (anonymous: CHARACTER; anonymous: STRING)
--                                       return STRING;
--  function "&"  (anonymous: CHARACTER; anonymous: CHARACTER)
--                                       return STRING;

--  function MINIMUM (L, R: STRING) return STRING;
--  function MAXIMUM (L, R: STRING) return STRING;

--  function MINIMUM (L: STRING) return CHARACTER;
--  function MAXIMUM (L: STRING) return CHARACTER;

type BOOLEAN_VECTOR is array (NATURAL range <>) of BOOLEAN;

--  The predefined operations for this type are as follows:
```

```
--  function "and"  (anonymous, anonymous: BOOLEAN_VECTOR)
--                                         return BOOLEAN_VECTOR;
--  function "or"   (anonymous, anonymous: BOOLEAN_VECTOR)
--                                         return BOOLEAN_VECTOR;
--  function "nand" (anonymous, anonymous: BOOLEAN_VECTOR)
--                                         return BOOLEAN_VECTOR;
--  function "nor"  (anonymous, anonymous: BOOLEAN_VECTOR)
--                                         return BOOLEAN_VECTOR;
--  function "xor"  (anonymous, anonymous: BOOLEAN_VECTOR)
--                                         return BOOLEAN_VECTOR;
--  function "xnor" (anonymous, anonymous: BOOLEAN_VECTOR)
--                                         return BOOLEAN_VECTOR;

--  function "not"  (anonymous: BOOLEAN_VECTOR)
--                                         return BOOLEAN_VECTOR;

--  function "and"  (anonymous: BOOLEAN_VECTOR; anonymous: BOOLEAN)
--                                         return BOOLEAN_VECTOR;
--  function "and"  (anonymous: BOOLEAN; anonymous: BOOLEAN_VECTOR)
--                                         return BOOLEAN_VECTOR;
--  function "or"   (anonymous: BOOLEAN_VECTOR; anonymous: BOOLEAN)
--                                         return BOOLEAN_VECTOR;
--  function "or"   (anonymous: BOOLEAN; anonymous : BOOLEAN_VECTOR)
--                                         return BOOLEAN_VECTOR;
--  function "nand" (anonymous: BOOLEAN_VECTOR; anonymous: BOOLEAN)
--                                         return BOOLEAN_VECTOR;
--  function "nand" (anonymous: BOOLEAN; anonymous: BOOLEAN_VECTOR)
--                                         return BOOLEAN_VECTOR;
--  function "nor"  (anonymous: BOOLEAN_VECTOR; anonymous: BOOLEAN)
--                                         return BOOLEAN_VECTOR;
--  function "nor"  (anonymous: BOOLEAN; anonymous: BOOLEAN_VECTOR)
--                                         return BOOLEAN_VECTOR;
--  function "xor"  (anonymous: BOOLEAN_VECTOR; anonymous: BOOLEAN)
--                                         return BOOLEAN_VECTOR;
--  function "xor"  (anonymous: BOOLEAN; anonymous: BOOLEAN_VECTOR)
--                                         return BOOLEAN_VECTOR;
--  function "xnor" (anonymous: BOOLEAN_VECTOR; anonymous: BOOLEAN)
--                                         return BOOLEAN_VECTOR;
--  function "xnor" (anonymous: BOOLEAN; anonymous: BOOLEAN_VECTOR)
--                                         return BOOLEAN_VECTOR;

--  function "and"  (anonymous: BOOLEAN_VECTOR) return BOOLEAN;
--  function "or"   (anonymous: BOOLEAN_VECTOR) return BOOLEAN;
--  function "nand" (anonymous: BOOLEAN_VECTOR) return BOOLEAN;
--  function "nor"  (anonymous: BOOLEAN_VECTOR) return BOOLEAN;
--  function "xor"  (anonymous: BOOLEAN_VECTOR) return BOOLEAN;
--  function "xnor" (anonymous: BOOLEAN_VECTOR) return BOOLEAN;

--  function "sll"  (anonymous: BOOLEAN_VECTOR; anonymous: INTEGER)
--                                         return BOOLEAN_VECTOR;
--  function "srl"  (anonymous: BOOLEAN_VECTOR; anonymous: INTEGER)
--                                         return BOOLEAN_VECTOR;
--  function "sla"  (anonymous: BOOLEAN_VECTOR; anonymous: INTEGER)
--                                         return BOOLEAN_VECTOR;
```

```
--  function "sra"  (anonymous: BOOLEAN_VECTOR; anonymous: INTEGER)
--                                          return BOOLEAN_VECTOR;
--  function "rol"  (anonymous: BOOLEAN_VECTOR; anonymous: INTEGER)
--                                          return BOOLEAN_VECTOR;
--  function "ror"  (anonymous: BOOLEAN_VECTOR; anonymous: INTEGER)
--                                          return BOOLEAN_VECTOR;

--  function "="    (anonymous, anonymous: BOOLEAN_VECTOR)
--                                          return BOOLEAN;
--  function "/="   (anonymous, anonymous: BOOLEAN_VECTOR)
--                                          return BOOLEAN;
--  function "<"    (anonymous, anonymous: BOOLEAN_VECTOR)
--                                          return BOOLEAN;
--  function "<="   (anonymous, anonymous: BOOLEAN_VECTOR)
--                                          return BOOLEAN;
--  function ">"    (anonymous, anonymous: BOOLEAN_VECTOR)
--                                          return BOOLEAN;
--  function ">="   (anonymous, anonymous: BOOLEAN_VECTOR)
--                                          return BOOLEAN;

--  function "?="   (anonymous, anonymous: BOOLEAN_VECTOR)
--                                          return BOOLEAN;
--  function "?/="  (anonymous, anonymous: BOOLEAN_VECTOR)
--                                          return BOOLEAN;

--  function "&"    (anonymous: BOOLEAN_VECTOR;
--                   anonymous: BOOLEAN_VECTOR)
--                                          return BOOLEAN_VECTOR;
--  function "&"    (anonymous: BOOLEAN_VECTOR; anonymous: BOOLEAN)
--                                          return BOOLEAN_VECTOR;
--  function "&"    (anonymous: BOOLEAN; anonymous: BOOLEAN_VECTOR)
--                                          return BOOLEAN_VECTOR;
--  function "&"    (anonymous: BOOLEAN; anonymous: BOOLEAN)
--                                          return BOOLEAN_VECTOR;

--  function MINIMUM (L, R: BOOLEAN_VECTOR) return BOOLEAN_VECTOR;
--  function MAXIMUM (L, R: BOOLEAN_VECTOR) return BOOLEAN_VECTOR;

--  function MINIMUM (L: BOOLEAN_VECTOR) return BOOLEAN;
--  function MAXIMUM (L: BOOLEAN_VECTOR) return BOOLEAN;

type BIT_VECTOR is array (NATURAL range <>) of BIT;

--  The predefined operations for this type are as follows:

--  function "and"  (anonymous, anonymous: BIT_VECTOR)
--                                           return BIT_VECTOR;
--  function "or"   (anonymous, anonymous: BIT_VECTOR)
--                                           return BIT_VECTOR;
--  function "nand" (anonymous, anonymous: BIT_VECTOR)
--                                           return BIT_VECTOR;
--  function "nor"  (anonymous, anonymous: BIT_VECTOR)
--                                           return BIT_VECTOR;
--  function "xor"  (anonymous, anonymous: BIT_VECTOR)
```

```
--                                              return BIT_VECTOR;
--  function "xnor" (anonymous, anonymous: BIT_VECTOR)
--                                              return BIT_VECTOR;

--  function "not"  (anonymous: BIT_VECTOR) return BIT_VECTOR;

--  function "and"  (anonymous: BIT_VECTOR; anonymous : BIT)
--                                              return BIT_VECTOR;
--  function "and"  (anonymous: BIT; anonymous : BIT_VECTOR)
--                                              return BIT_VECTOR;
--  function "or"   (anonymous: BIT_VECTOR; anonymous : BIT)
--                                              return BIT_VECTOR;
--  function "or"   (anonymous: BIT; anonymous : BIT_VECTOR)
--                                              return BIT_VECTOR;
--  function "nand" (anonymous: BIT_VECTOR; anonymous : BIT)
--                                              return BIT_VECTOR;
--  function "nand" (anonymous: BIT; anonymous : BIT_VECTOR)
--                                              return BIT_VECTOR;
--  function "nor"  (anonymous: BIT_VECTOR; anonymous : BIT)
--                                              return BIT_VECTOR;
--  function "nor"  (anonymous: BIT; anonymous : BIT_VECTOR)
--                                              return BIT_VECTOR;
--  function "xor"  (anonymous: BIT_VECTOR; anonymous : BIT)
--                                              return BIT_VECTOR;
--  function "xor"  (anonymous: BIT; anonymous : BIT_VECTOR)
--                                              return BIT_VECTOR;
--  function "xnor" (anonymous: BIT_VECTOR; anonymous : BIT)
--                                              return BIT_VECTOR;
--  function "xnor" (anonymous: BIT; anonymous : BIT_VECTOR)
--                                              return BIT_VECTOR;

--  function "and"  (anonymous: BIT_VECTOR) return BIT;
--  function "or"   (anonymous: BIT_VECTOR) return BIT;
--  function "nand" (anonymous: BIT_VECTOR) return BIT;
--  function "nor"  (anonymous: BIT_VECTOR) return BIT;
--  function "xor"  (anonymous: BIT_VECTOR) return BIT;
--  function "xnor" (anonymous: BIT_VECTOR) return BIT;

--  function "sll"  (anonymous: BIT_VECTOR; anonymous: INTEGER)
--                                              return BIT_VECTOR;
--  function "srl"  (anonymous: BIT_VECTOR; anonymous: INTEGER)
--                                              return BIT_VECTOR;
--  function "sla"  (anonymous: BIT_VECTOR; anonymous: INTEGER)
--                                              return BIT_VECTOR;
--  function "sra"  (anonymous: BIT_VECTOR; anonymous: INTEGER)
--                                              return BIT_VECTOR;
--  function "rol"  (anonymous: BIT_VECTOR; anonymous: INTEGER)
--                                              return BIT_VECTOR;
--  function "ror"  (anonymous: BIT_VECTOR; anonymous: INTEGER)
--                                              return BIT_VECTOR;

--  function "="    (anonymous, anonymous: BIT_VECTOR)
--                                              return BOOLEAN;
--  function "/="   (anonymous, anonymous: BIT_VECTOR)
```

```
--                                          return BOOLEAN;
--  function "<"    (anonymous, anonymous: BIT_VECTOR)
--                                          return BOOLEAN;
--  function "<="   (anonymous, anonymous: BIT_VECTOR)
--                                          return BOOLEAN;
--  function ">"    (anonymous, anonymous: BIT_VECTOR)
--                                          return BOOLEAN;
--  function ">="   (anonymous, anonymous: BIT_VECTOR)
--                                          return BOOLEAN;

--  function "?="   (anonymous, anonymous: BIT_VECTOR) return BIT;
--  function "?/="  (anonymous, anonymous: BIT_VECTOR) return BIT;

--  function "&"    (anonymous: BIT_VECTOR; anonymous: BIT_VECTOR)
--                                          return BIT_VECTOR;
--  function "&"    (anonymous: BIT_VECTOR; anonymous: BIT)
--                                          return BIT_VECTOR;
--  function "&"    (anonymous: BIT; anonymous: BIT_VECTOR)
--                                          return BIT_VECTOR;
--  function "&"    (anonymous: BIT; anonymous: BIT)
--                                          return BIT_VECTOR;

--  function MINIMUM (L, R: BIT_VECTOR) return BIT_VECTOR;
--  function MAXIMUM (L, R: BIT_VECTOR) return BIT_VECTOR;

--  function MINIMUM (L: BIT_VECTOR) return BIT;
--  function MAXIMUM (L: BIT_VECTOR) return BIT;

--  function TO_STRING (VALUE: BIT_VECTOR) return STRING;

--  alias    TO_BSTRING       is TO_STRING
--                                [BIT_VECTOR return STRING];
--  alias    TO_BINARY_STRING is TO_STRING
--                                [BIT_VECTOR return STRING];
--  function TO_OSTRING (VALUE: BIT_VECTOR) return STRING;
--  alias    TO_OCTAL_STRING  is TO_OSTRING
--                                [BIT_VECTOR return STRING];
--  function TO_HSTRING (VALUE: BIT_VECTOR) return STRING;
--  alias    TO_HEX_STRING    is TO_HSTRING
--                                [BIT_VECTOR return STRING];

type INTEGER_VECTOR is array (NATURAL range <>) of INTEGER;

--  The predefined operations for this type are as follows:

--  function "="  (anonymous, anonymous: INTEGER_VECTOR)
--                                          return BOOLEAN;
--  function "/=" (anonymous, anonymous: INTEGER_VECTOR)
--                                          return BOOLEAN;
--  function "<"  (anonymous, anonymous: INTEGER_VECTOR)
--                                          return BOOLEAN;
--  function "<=" (anonymous, anonymous: INTEGER_VECTOR)
--                                          return BOOLEAN;
--  function ">"  (anonymous, anonymous: INTEGER_VECTOR)
```

```
--                                          return BOOLEAN;
--  function ">=" (anonymous, anonymous: INTEGER_VECTOR)
--                                          return BOOLEAN;

--  function "&"  (anonymous: INTEGER_VECTOR;
--                 anonymous: INTEGER_VECTOR) return INTEGER_VECTOR;
--  function "&"  (anonymous: INTEGER_VECTOR;
--                 anonymous: INTEGER)        return INTEGER_VECTOR;
--  function "&"  (anonymous: INTEGER;
--                 anonymous: INTEGER_VECTOR) return INTEGER_VECTOR;
--  function "&"  (anonymous: INTEGER;
--                 anonymous: INTEGER)        return INTEGER_VECTOR;

--  function MINIMUM (L, R: INTEGER_VECTOR) return INTEGER_VECTOR;
--  function MAXIMUM (L, R: INTEGER_VECTOR) return INTEGER_VECTOR;

--  function MINIMUM (L: INTEGER_VECTOR) return INTEGER;
--  function MAXIMUM (L: INTEGER_VECTOR) return INTEGER;

type REAL_VECTOR is array (NATURAL range <>) of REAL;

--  The predefined operations for this type are as follows:

--  function "="  (anonymous, anonymous: REAL_VECTOR)
--                                          return BOOLEAN;
--  function "/=" (anonymous, anonymous: REAL_VECTOR)
--                                          return BOOLEAN;

--  function "&"  (anonymous: REAL_VECTOR; anonymous: REAL_VECTOR)
--                                          return REAL_VECTOR;
--  function "&"  (anonymous: REAL_VECTOR; anonymous: REAL)
--                                          return REAL_VECTOR;
--  function "&"  (anonymous: REAL; anonymous: REAL_VECTOR)
--                                          return REAL_VECTOR;
--  function "&"  (anonymous: REAL; anonymous: REAL)
--                                          return REAL_VECTOR;

--  function MINIMUM (L: REAL_VECTOR) return REAL;
--  function MAXIMUM (L: REAL_VECTOR) return REAL;

type TIME_VECTOR is array (NATURAL range <>) of TIME;

--  The predefined operations for this type are as follows:

--  function "="  (anonymous, anonymous: TIME_VECTOR)
--                                          return BOOLEAN;
--  function "/=" (anonymous, anonymous: TIME_VECTOR)
--                                          return BOOLEAN;

--  function "&"  (anonymous: TIME_VECTOR; anonymous: TIME_VECTOR)
--                                          return TIME_VECTOR;
--  function "&"  (anonymous: TIME_VECTOR; anonymous: TIME)
--                                          return TIME_VECTOR;
--  function "&"  (anonymous: TIME; anonymous: TIME_VECTOR)
```

```
--                                          return TIME_VECTOR;
--  function "&"  (anonymous: TIME; anonymous: TIME)
--                                          return TIME_VECTOR;

--  function MINIMUM (L: TIME_VECTOR) return TIME;
--  function MAXIMUM (L: TIME_VECTOR) return TIME;

--  The predefined types for opening files:

type FILE_OPEN_KIND is (
   READ_MODE,              --  Resulting access mode is read-only.
   WRITE_MODE,             --  Resulting access mode is write-only.
   APPEND_MODE);           --  Resulting access mode is write-only;
                           --  information is appended to the end
                           --  of the existing file.

--  The predefined operations for this type are as follows:

--  function "="  (anonymous, anonymous: FILE_OPEN_KIND)
--                                          return BOOLEAN;
--  function "/=" (anonymous, anonymous: FILE_OPEN_KIND)
--                                          return BOOLEAN;
--  function "<"  (anonymous, anonymous: FILE_OPEN_KIND)
--                                          return BOOLEAN;
--  function "<=" (anonymous, anonymous: FILE_OPEN_KIND)
--                                          return BOOLEAN;
--  function ">"  (anonymous, anonymous: FILE_OPEN_KIND)
--                                          return BOOLEAN;
--  function ">=" (anonymous, anonymous: FILE_OPEN_KIND)
--                                          return BOOLEAN;

--  function MINIMUM (L, R: FILE_OPEN_KIND) return FILE_OPEN_KIND;
--  function MAXIMUM (L, R: FILE_OPEN_KIND) return FILE_OPEN_KIND;

type FILE_OPEN_STATUS is (
   OPEN_OK,                --  File open was successful.
   STATUS_ERROR,           --  File object was already open.
   NAME_ERROR,             --  External file not found
                           --  or inaccessible.
   MODE_ERROR);            --  Could not open file with requested
                           --  access mode.

--  The predefined operations for this type are as follows:

--  function "="  (anonymous, anonymous: FILE_OPEN_STATUS)
--                                          return BOOLEAN;
--  function "/=" (anonymous, anonymous: FILE_OPEN_STATUS)
--                                          return BOOLEAN;
--  function "<"  (anonymous, anonymous: FILE_OPEN_STATUS)
--                                          return BOOLEAN;
--  function "<=" (anonymous, anonymous: FILE_OPEN_STATUS)
--                                          return BOOLEAN;
--  function ">"  (anonymous, anonymous: FILE_OPEN_STATUS)
--                                          return BOOLEAN;
```

```
    --  function ">=" (anonymous, anonymous: FILE_OPEN_STATUS)
    --                                          return BOOLEAN;

    --  function MINIMUM (L, R: FILE_OPEN_STATUS)
                                                return FILE_OPEN_STATUS;
    --  function MAXIMUM (L, R: FILE_OPEN_STATUS)
                                                return FILE_OPEN_STATUS;

    --  The 'FOREIGN attribute:

    attribute FOREIGN: STRING;

    --  Predefined TO_STRING operations on scalar types

    --  function TO_STRING (VALUE: BOOLEAN)           return STRING;
    --  function TO_STRING (VALUE: BIT)               return STRING;
    --  function TO_STRING (VALUE: CHARACTER)         return STRING;
    --  function TO_STRING (VALUE: SEVERITY_LEVEL)    return STRING;
    --  function TO_STRING (VALUE: universal_integer) return STRING;
    --  function TO_STRING (VALUE: universal_real)    return STRING;
    --  function TO_STRING (VALUE: INTEGER)           return STRING;
    --  function TO_STRING (VALUE: REAL)              return STRING;
    --  function TO_STRING (VALUE: TIME)              return STRING;
    --  function TO_STRING (VALUE: FILE_OPEN_KIND)    return STRING;
    --  function TO_STRING (VALUE: FILE_OPEN_STATUS)  return STRING;

    --  Predefined overloaded TO_STRING operations

    --  function TO_STRING (VALUE: REAL; DIGITS: NATURAL)
                                                  return STRING;
    --  function TO_STRING (VALUE: REAL; FORMAT: STRING)
                                                  return STRING;
    --  function TO_STRING (VALUE: TIME; UNIT: TIME)  return STRING;

end STANDARD;
```

The 'FOREIGN attribute shall be associated only with architectures (see 3.3) or with subprograms. In the latter case, the attribute specification shall appear in the declarative part in which the subprogram is declared (see 4.2).

NOTE 1—The ASCII mnemonics for file separator (FS), group separator (GS), record separator (RS), and unit separator (US) are represented by FSP, GSP, RSP, and USP, respectively, in type CHARACTER in order to avoid conflict with the units of type TIME.

NOTE 2—The declarative parts and statement parts of design entities whose corresponding architectures are decorated with the 'FOREIGN attribute and subprograms that are likewise decorated are subject to special elaboration rules. See 14.4.1 and 14.5.1.

16.4 Package TEXTIO

Package TEXTIO contains declarations of types and subprograms that support formatted I/O operations on text files.

```
package TEXTIO is
```



```
--  Type definitions for text I/O:


type LINE is access STRING;  --  A LINE is a pointer
                             --  to a STRING value.

--  The predefined operations for this type are as follows:

--  function"="  (anonymous, anonymous: LINE) return BOOLEAN;
--  function"/=" (anonymous, anonymous: LINE) return BOOLEAN;

--  procedure DEALLOCATE (P: inout LINE);

type TEXT is file of STRING;  --  A file of variable-length
                              --  ASCII records.

--  The predefined operations for this type are as follows:

--  procedure FILE_OPEN  (file F: TEXT; External_Name; in STRING;
--                        Open_Kind: in FILE_OPEN_KIND := READ_MODE);
--  procedure FILE_OPEN  (Status: out FILE_OPEN_STATUS; file F: TEXT;
--                        External_Name: in STRING;
--                        Open_Kind: in FILE_OPEN_KIND := READ_MODE);
--  procedure FILE_CLOSE (file F: TEXT);
--  procedure READ       (file F: TEXT; VALUE: out STRING);
--  procedure WRITE      (file F: TEXT; VALUE: in STRING);
--  procedure FLUSH      (file F: TEXT);
--  function  ENDFILE    (file F: TEXT) return BOOLEAN;

type SIDE is (RIGHT, LEFT);  --  For justifying output data
                             --  within fields.

--  The predefined operations for this type are as follows:

--  function "="  (anonymous, anonymous: SIDE) return BOOLEAN;
--  function "/=" (anonymous, anonymous: SIDE) return BOOLEAN;
--  function "<"  (anonymous, anonymous: SIDE) return BOOLEAN;
--  function "<=" (anonymous, anonymous: SIDE) return BOOLEAN;
--  function ">"  (anonymous, anonymous: SIDE) return BOOLEAN;
--  function ">=" (anonymous, anonymous: SIDE) return BOOLEAN;

--  function MINIMUM (L, R: SIDE) return SIDE;
--  function MAXIMUM (L, R: SIDE) return SIDE;

--  function TO_STRING (VALUE: SIDE) return STRING;

subtype WIDTH is NATURAL;  --  For specifying widths of output fields.

function JUSTIFY (VALUE: STRING;
                  JUSTIFIED: SIDE := RIGHT;
                  FIELD: WIDTH := 0 ) return STRING;

--  Standard text files:
```

```
file INPUT:  TEXT open READ_MODE  is "STD_INPUT";

file OUTPUT: TEXT open WRITE_MODE is "STD_OUTPUT";

--  Input routines for standard types:

procedure READLINE (file F: TEXT; L: inout LINE);

procedure READ (L: inout LINE; VALUE: out BIT;
                               GOOD:  out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out BIT);

procedure READ (L: inout LINE; VALUE: out BIT_VECTOR;
                               GOOD:  out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out BIT_VECTOR);

procedure READ (L: inout LINE; VALUE: out BOOLEAN;
                               GOOD:  out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out BOOLEAN);

procedure READ (L: inout LINE; VALUE: out CHARACTER;
                               GOOD:  out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out CHARACTER);

procedure READ (L: inout LINE; VALUE: out INTEGER;
                               GOOD:  out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out INTEGER);

procedure READ (L: inout LINE; VALUE: out REAL;
                               GOOD:  out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out REAL);

procedure READ (L: inout LINE; VALUE: out STRING;
                               GOOD:  out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out STRING);

procedure READ (L: inout LINE; VALUE: out TIME;
                               GOOD:  out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out TIME);

procedure SREAD (L: inout LINE; VALUE:  out STRING;
                                STRLEN: out NATURAL);
alias STRING_READ is SREAD [LINE, STRING, NATURAL];

alias BREAD is READ [LINE, BIT_VECTOR, BOOLEAN];
alias BREAD is READ [LINE, BIT_VECTOR];
alias BINARY_READ is READ [LINE, BIT_VECTOR, BOOLEAN];
alias BINARY_READ is READ [LINE, BIT_VECTOR];

procedure OREAD (L: inout LINE; VALUE: out BIT_VECTOR;
                                GOOD:  out BOOLEAN);
procedure OREAD (L: inout LINE; VALUE: out BIT_VECTOR);
alias OCTAL_READ is OREAD [LINE, BIT_VECTOR, BOOLEAN];
alias OCTAL_READ is OREAD [LINE, BIT_VECTOR];
```

```
procedure HREAD (L: inout LINE; VALUE: out BIT_VECTOR;
                                 GOOD:  out BOOLEAN);
procedure HREAD (L: inout LINE; VALUE: out BIT_VECTOR);
alias HEX_READ is HREAD [LINE, BIT_VECTOR, BOOLEAN];
alias HEX_READ is HREAD [LINE, BIT_VECTOR];

--  Output routines for standard types:

procedure WRITELINE (file F: TEXT; L: inout LINE);

procedure TEE   (file F: TEXT; L: inout LINE);

procedure WRITE (L: inout LINE; VALUE: in BIT;
                 JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);

procedure WRITE (L: inout LINE; VALUE: in BIT_VECTOR;
                 JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);

procedure WRITE (L: inout LINE; VALUE: in BOOLEAN;
                 JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);

procedure WRITE (L: inout LINE; VALUE: in CHARACTER;
                 JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);

procedure WRITE (L: inout LINE; VALUE: in INTEGER;
                 JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);

procedure WRITE (L: inout LINE; VALUE: in REAL;
                 JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0;
                 DIGITS: in NATURAL:= 0);

procedure WRITE (L: inout LINE; VALUE: in REAL;
                 FORMAT: in STRING);

procedure WRITE (L: inout LINE; VALUE: in STRING;
                 JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);

procedure WRITE (L: inout LINE; VALUE: in TIME;
                 JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0;
                 UNIT: in TIME:= ns);

alias SWRITE       is WRITE [LINE, STRING, SIDE, WIDTH];
alias STRING_WRITE is WRITE [LINE, STRING, SIDE, WIDTH];

alias BWRITE       is WRITE [LINE, BIT_VECTOR, SIDE, WIDTH];
alias BINARY_WRITE is WRITE [LINE, BIT_VECTOR, SIDE, WIDTH];

procedure OWRITE (L: inout LINE; VALUE: in BIT_VECTOR;
                  JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
alias OCTAL_WRITE is OWRITE [LINE, BIT_VECTOR, SIDE, WIDTH];

procedure HWRITE (L: inout LINE; VALUE: in BIT_VECTOR;
                  JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
```

```
    alias HEX_WRITE is HWRITE [LINE, BIT_VECTOR, SIDE, WIDTH];

end TEXTIO;
```

Procedures READLINE, WRITELINE, and TEE declared in package TEXTIO read and write entire lines of a file of type TEXT. Procedure READLINE causes the next line to be read from the file and returns as the value of parameter L an access value that designates an object representing that line. If parameter L contains a non-null access value at the start of the call, the procedure may deallocate the object designated by that value. The representation of the line does not contain the representation of the end of the line. It is an error if the file specified in a call to READLINE is not open or, if open, the file has an access mode other than read-only (see 5.5.2). Procedures WRITELINE and TEE each cause the current line designated by parameter L to be written to the file and returns with the value of parameter L designating a null string. Procedure TEE additionally causes the current line to be written to the file OUTPUT. If parameter L contains a null access value at the start of the call, then a null string is written to the file or files. If parameter L contains a non-null access value at the start of the call, the procedures may deallocate the object designated by that value. It is an error if the file specified in a call to WRITELINE or TEE is not open or, if open, the file has an access mode other than write-only.

The language does not define the representation of the end of a line. An implementation shall allow all possible values of types CHARACTER and STRING to be written to a file. However, as an implementation is permitted to use certain values of types CHARACTER and STRING as line delimiters, it might not be possible to read these values from a TEXT file.

A line feed (LF) format effector occurring as an element of a string written to a file of type TEXT, either using procedure WRITELINE or TEE, or using the WRITE operation implicitly defined for the type TEXT, is interpreted by the implementation as signifying the end of a line. The implementation shall transform the LF into the implementation-defined representation of the end of a line.

The JUSTIFY operation formats a string value within a *field* that is at least as long as required to contain the value. Parameter FIELD specifies the desired field width. Since the actual field width will always be at least large enough to hold the string value, the default value 0 for the FIELD parameter has the effect of causing the string value to be contained in a field of exactly the right width (i.e., no additional leading or trailing spaces). Parameter JUSTIFIED specifies whether the string value is to be right- or left-justified within the field; the default is right-justified. If the FIELD parameter describes a field width larger than the number of characters in the string value, space characters are used to fill the remaining characters in the field.

Each READ, SREAD, OREAD, and HREAD procedure declared in package TEXTIO extracts data from the beginning of the string value designated by parameter L and modifies the value so that it designates the remaining portion of the line on exit. Each procedure may modify the value of the object designated by the parameter L at the start of the call or may deallocate the object.

The READ procedures defined for a given type other than CHARACTER and STRING begin by skipping leading *whitespace characters*. A whitespace character is defined as a space, a nonbreaking space, or a horizontal tabulation character (SP, NBSP, or HT). For all READ procedures, characters are then removed from L and composed into a string representation (see 5.7) of the value of the specified type. The READ procedure for type BIT_VECTOR also removes underline characters from L, provided the underline character does not precede the string representation of the value and does not immediately follow another underline character. The removed underline characters are not added to the string representation. For all READ procedures, character removal and string composition stops when the end of the line is encountered. Character removal and string composition also stops when a character is encountered that cannot be part of the value according to the rules for string representations, or, in the case of the READ procedure for BIT_VECTOR, is not an underline character that can be removed according to the preceding rule; this character is not removed from L and is not added to the string representation of the value. The READ procedures for types STRING and BIT_VECTOR also terminate acceptance when VALUE'LENGTH

characters have been accepted (not counting underline characters in the case of the READ procedure for BIT_VECTOR). Again using the rules of 5.7, the accepted characters are then interpreted as a string representation of the specified type. The READ does not succeed if the sequence of characters composed into the string representation is not a valid string representation of a value of the specified type or, in the case of types STRING and BIT_VECTOR, if the sequence does not contain VALUE'LENGTH characters.

The SREAD procedure begins by skipping leading whitespace characters. Characters are then removed and composed from left to right into a string provided as the VALUE parameter. Character removal and string composition stops when the end of the line is encountered. Character removal and string composition also stops when a whitespace character is encountered or VALUE'LENGTH characters have been accepted; the whitespace character is not removed from L and is not added to the string. The number of characters composed into the string is provided as the value of the STRLEN parameter. The values of elements of the string to the right of those composed by the SREAD procedure are not defined by this standard.

The OREAD and HREAD procedures begin by skipping leading whitespace characters. Characters are then removed and composed into a sequence of octal (respectively, hexadecimal) digits. Each underline character is also removed from L, provided the underline character does not precede the sequence of octal (respectively, hexadecimal) digits and does not immediately follow another underline character. The removed underline characters are not added to the string representation. Character removal and composition stops when the end of the line is encountered. Character removal and string composition also stops when a character is encountered that is not an octal (respectively, hexadecimal) digit or an underline character that can be removed according to the preceding rule; this character is not removed from L and is not added to the string. Moreover, character removal and composition stops when the expected number of digits have been removed, where the expected number of digits is the smallest integer greater than or equal to VALUE'LENGTH divided by three (respectively, four). The OREAD or HREAD procedure does not succeed if less than the expected number of digits are removed. Otherwise, the sequence of octal (respectively, hexadecimal) digits is interpreted as an octal (respectively, hexadecimal) number and converted into a binary number of three (respectively, four) times VALUE'LENGTH bits. The rightmost VALUE'LENGTH bits of the binary number are used to form the result for the VALUE parameter, with a '0' element corresponding to a 0 bit and a '1' element corresponding to a 1 bit. The OREAD or HREAD procedure does not succeed if any unused bits are 1.

Each WRITE procedure similarly appends data to the end of the string value designated by parameter L. The format of the appended data is defined by the string representations defined in 5.7.

The OWRITE and HWRITE procedures append the octal (respectively, hexadecimal) representation of the VALUE parameter to the end of the string value designated by parameter L. The octal (respectively, hexadecimal) representation is the value given by application of the TO_OSTRING (respectively, TO_HSTRING) operation to the VALUE parameter (see 5.3.2.4).

For each WRITE, OWRITE, and HWRITE procedure, after data is appended to the string value designated by the parameter L, L designates the entire line. The procedure may modify the value of the object designated by the parameter L at the start of the call or may deallocate the object.

The READ and WRITE procedures for the types BIT_VECTOR and STRING respectively read and write the element values in left-to-right order.

For each predefined data type there are two READ procedures declared in package TEXTIO. The first has three parameters: L, the line to read from; VALUE, the value read from the line; and GOOD, a Boolean flag that indicates whether the read operation succeeded or not. For example, the operation READ (L, IntVal, OK) would return with OK set to FALSE, L unchanged, and IntVal undefined if IntVal is a variable of type INTEGER and L designates the line “ABC.” The success indication returned via parameter GOOD allows a process to recover gracefully from unexpected discrepancies in input format. The second form of read operation has only the parameters L and VALUE. If the requested type cannot be read into VALUE from

line L, then an error occurs. Thus, the operation READ (L, IntVal) would cause an error to occur if IntVal is of type INTEGER and L designates the line "ABC". For the predefined type BIT_VECTOR, there are likewise two OREAD and two HREAD procedures, with similar parameters.

For each predefined data type there is one or more WRITE procedure declared in package TEXTIO. Each of these has at least two parameters: L, the line to which to write, and VALUE, the value to be written. The additional parameters JUSTIFIED, FIELD, DIGITS, FORMAT, and UNIT control the formatting of output data. Each write operation appends data to a line formatted within a *field* that is at least as long as required to represent the data value. Parameters FIELD and JUSTIFIED specify the desired field width and justification, as for the JUSTIFY operation. For the predefined type BIT_VECTOR, there is likewise one OWRITE and one HWRITE procedure, with similar parameters.

Parameter DIGITS specifies how many digits to the right of the decimal point are to be output when writing a real number; the default value 0 indicates that the number should be output in standard form, consisting of a normalized mantissa plus exponent (e.g., 1.079236e-23). If DIGITS is non-zero, then the real number is output as an integer part followed by '.' followed by the fractional part, using the specified number of digits (e.g., 3.14159).

Parameter FORMAT specifies how values of type REAL are to be formatted. The formatting is determined in the same manner as for the TO_STRING operation for type REAL with the FORMAT parameter (see 5.2.6).

Parameter UNIT specifies how values of type TIME are to be formatted. The value of this parameter shall be equal to one of the units declared as part of the declaration of type TIME; the result is that the TIME value is formatted as an integer or real literal representing the number of multiples of this unit, followed by the name of the unit itself. The name of the unit is formatted using only lowercase characters. Thus the procedure call WRITE(Line, 5 ns, UNIT=>us) would result in the string value "0.005 us" being appended to the string value designated by Line, whereas WRITE(Line, 5 ns) would result in the string value "5 ns" being appended (since the default UNIT value is ns).

Function ENDFILE is defined for files of type TEXT by the implicit declaration of that function as part of the declaration of the file type.

NOTE 1—For a variable L of type Line, attribute L'Length gives the current length of the line, whether that line is being read or written. For a line L that is being written, the value of L'Length gives the number of characters that have already been written to the line; this is equivalent to the column number of the last character of the line. For a line L that is being read, the value of L'Length gives the number of characters on that line remaining to be read. In particular, the expression L'Length = 0 is true precisely when the end of the current line has been reached.

NOTE 2—Since the execution of a read or write operation may modify or deallocate the string object designated by input parameter L of type Line for that operation, a dangling reference may result if the value of a variable L of type Line is assigned to another access variable and then a read or write operation is performed on L.

NOTE 3—A call to a WRITE procedure with a string literal for the VALUE parameter is ambiguous, as the string could be interpreted as a value of type STRING or type BIT_VECTOR. If the intention is to write a value of type STRING, the alias SWRITE can be called without ambiguity.

16.5 Standard environment package

Package ENV contains declarations that provide a VHDL interface to the host environment.

```
package ENV is

   procedure STOP (STATUS: INTEGER);
   procedure STOP;
```



```
    procedure FINISH (STATUS: INTEGER);
    procedure FINISH;

    function RESOLUTION_LIMIT return DELAY_LENGTH;

end package ENV;
```

Execution of the STOP procedures causes the same action by the host simulator as that caused by the `vhpi_control` function called with the argument `vhpiStop` (see 23.5). Execution of the FINISH procedures causes the same action by the host simulator as that caused by the `vhpi_control` function called with the argument `vhpiFinish` (see 23.5). Execution shall not return to the VHDL description after a call to the FINISH procedure. For the procedures with the STATUS parameter, the value of the STATUS parameter may be used in an implementation defined manner by the host simulator. For the procedures with no parameter, the effect is the same as that caused by the `vhpi_control` function with no additional arguments beyond the `vhpiStop` or `vhpiFinish` argument.

The function RESOLUTION_LIMIT returns the value of the resolution limit (see 5.2.4.2).

NOTE 1—The value of the STATUS parameter of the STOP and FINISH procedures may, for example, be provided to a simulation control script for use in determining what external control actions to perform.

NOTE 2—An implementation shall provide the STOP and FINISH procedures in package ENV regardless of whether it implements the VHPI.

NOTE 3—A description may include a comparison of the resolution limit with a literal of type TIME, but an error occurs if the literal includes a unit that is smaller than the resolution limit (see 5.2.4.2). For example, the expression "RESOLUTION_LIMIT <= ns" will cause an error if the resolution limit is greater than ns. The error can be avoided by using a literal with a suitably larger unit, for example, 1.0E–9 sec. Such a literal may be truncated to zero time units, but will not cause an error.

16.6 Standard mathematical packages

The library denoted by the library logical name IEEE contains packages MATH_REAL and MATH_COMPLEX. The following conformance rules shall apply as they pertain to the use and implementation of these packages:

a) The package declarations may be modified to include additional data required by tools, but modifications shall in no way change the external interfaces or simulation behavior of the description. It is permissible to add comments and/or attributes to the package declarations, but not to change or delete any original lines of the approved package declarations.

b) The standard mathematical definition and conventional meaning of the mathematical functions that are part of the packages, together with the MATH_REAL and MATH_COMPLEX package declarations, represent the formal semantics of the implementation of the MATH_REAL and MATH_COMPLEX packages. An implementation is provided as a guideline in the machine-readable files accompanying this standard (see Annex A). Implementors of these packages may choose to simply compile the package bodies provided in the files, or they may choose to implement the package bodies in the most efficient form available to them. Implementations should conform to the semantics and minimum precision required by this standard.

c) The MATH REAL package shall be built on top of the standard data type and precision requirements for floating-point operations defined in STD.STANDARD.

d) The minimum precision required is that specified by this standard for floating-point types (see 5.2.5.1). Because of this reason and the fact that the functions are implemented on digital computers with only finite precision, the functions and constants in this set of packages can, at best, only approximate the corresponding mathematically defined functions and constants. An implementation is allowed to provide a higher precision than the minimum required.

e) For some functions, the implementation shall deliver "prescribed results" for certain special arguments, as defined in the comments for the functions in the function declaration. The purpose is to strengthen the accuracy requirements at special argument values. Prescribed results take precedence over maximum relative error requirements.

f) The semantics of the standard require that all the functions in the packages detect and report invalid parameters (out of valid domain) through an assert statement. The domain of valid values is indicated in the MATH_REAL and MATH_COMPLEX package declarations. The default value of the severity level shall be ERROR.

g) The semantics of the standard do not require detection of overflow or underflow. Therefore, detection of underflow/overflow is optional and implementation dependent.

h) If an implementation chooses to provide any extensions to the packages beyond the minimum requirements of this standard (e.g., precision, overflow handling), then it shall document its behavior accordingly.

The declaration of each function includes the following information: description of the mathematical definition of the function; values to be returned by the function for special arguments; valid domain of values for the input arguments; error conditions; range of values into which the function maps the values in its domain; and notes on special accuracy situations, reachable values, usable domains, or algorithms to be used by an implementation.

The texts of the MATH_REAL and MATH_COMPLEX packages (both package declarations and package bodies) are included with this standard (see Annex A). Those texts are an official part of this standard.

NOTE—The mathematical packages were originally specified in IEEE Std 1076.2-1996 [B11]. The specifications in this standard supersede the original specifications.

16.7 Standard multivalue logic package

The library denoted by the library logical name IEEE contains packages STD_LOGIC_1164 and STD_LOGIC_TEXTIO.[11] The following conformance rules shall apply as they pertain to the use and implementation of this package:

a) The package declaration may be modified to include additional data required by tools, but modifications shall in no way change the external interfaces or simulation behavior of the description. It is permissible to add comments and/or attributes to the package declarations, but not to change or delete any original lines of the approved package declaration.

b) The STD_LOGIC_1164 package body provided in the machine-readable files accompanying this standard (see Annex A) represents the formal semantics of the implementation of the STD_LOGIC_1164 package declaration. Implementers of this package body may choose to simply compile the package body as it is; or they may choose to implement the package body in the most efficient form available to the user. Implementers shall not implement a semantic that differs from the formal semantic provided herein.

c) The STD_LOGIC_TEXTIO package is empty and is provided as a replacement for non-standard implementations of that package provided by implementers of previous versions of this standard. The declarations that appeared in those non-standard implementations appear in the package STD_LOGIC_1164 in this standard.

The text of the STD_LOGIC_1164 package (both package declaration and package body) and the STD_LOGIC_TEXTIO package (package declaration only) are included with this standard (see Annex A). That text is an official part of this standard.

[11]The package STD_LOGIC_TEXTIO was modified and used with permission of Synopsys, Inc. © 1990, 1991, and 1992.

NOTE—The name of the STD_LOGIC_1164 package derives from the fact that the package was originally specified in IEEE Std 1164-1993 [B16]. The specification in this standard supersedes the original specification.

16.8 Standard synthesis packages

NOTE—The specifications in this subclause were originally described in IEEE Std 1076.3-1997 [B12] The specifications in this standard supersede the original specifications.

16.8.1 Overview

16.8.1.1 Scope

This subclause defines standard practices for synthesizing binary digital electronic circuits from VHDL source code. It includes the following:

a) The hardware interpretation of values belonging to the BIT and BOOLEAN types defined in package STD.STANDARD and to the STD_ULOGIC type defined in package IEEE.STD_LOGIC_1164.

b) A function (STD_MATCH) that provides “don’t care” or “wild card” testing of values based on the STD_ULOGIC type.

c) Standard functions for representing sensitivity to the edge of a signal.

d) Packages that define one-dimensional array types for representing signed and unsigned arithmetic values, and that define arithmetic, shift, and type conversion operations on those types.

The packages are designed for use with this standard. Modifications that may be made to the packages for use with previous editions are described in 16.8.5.3.

Further related standard practices for synthesis of register-transfer level digital circuits are specified in IEEE Std 1076.6-2004 [B14].

16.8.1.2 Terminology

A *synthesis tool* is any tool that interprets VHDL source code as a description of an electronic circuit in accordance with the terms of this standard and derives an alternate description of that circuit. A synthesis tool is said to *accept* a VHDL construct if it allows that construct to be legal input; it is said to *interpret* the construct (or to provide an *interpretation* of the construct) by producing something that represents the construct. A synthesis tool is not required to provide an interpretation for every construct that it accepts, but only for those for which an interpretation is specified by this standard.

16.8.2 Interpretation of the standard logic types

16.8.2.1 General

This subclause (16.8.2) defines how a synthesis tool shall interpret values of the standard logic types defined in IEEE.STD_LOGIC_1164 and of the BIT and BOOLEAN types defined in STD.STANDARD. Simulation tools, however, shall continue to interpret these values according to the clauses of this standard in which the values are defined.

16.8.2.2 The STD_LOGIC_1164 values

The *logical values* '1', 'H', '0', and 'L' of type STD_ULOGIC are interpreted as representing one of two logic levels, where each logic level represents one of two distinct voltage ranges in the circuit to be synthesized.

The resolution function RESOLVED treats the values '0' and '1' as *forcing values* that override the *weak values* 'L' and 'H' when multiple sources drive the same signal.

The values 'U', 'X', 'W', and '–' are *metalogical values*; they define the behavior of the model itself rather than the behavior of the hardware being synthesized. The value 'U' represents the value of an object before it is explicitly assigned a value during simulation; the values 'X' and 'W' represent forcing and weak values, respectively, for which the model is not able to distinguish between logic levels.

The value '–' is also called the *don't care value*. This standard treats it in the same way as the other metalogical values except when it is furnished as an actual parameter to the STD_MATCH functions in the IEEE.NUMERIC_STD package or as an operand to a predefined matching relational operator (see 9.2.3). The STD_MATCH functions and the predefined matching relational operators use '–' to implement a "match all" or "wild card" matching.

The value 'Z' is called the *high-impedance value*, and represents the condition of a signal source when that source makes no effective contribution to the resolved value of the signal.

16.8.2.3 Static constant values

Wherever a synthesis tool accepts a reference to a locally static or globally static named constant, it shall treat that constant as the equivalent of the associated static expression.

16.8.2.4 Interpretation of logic values

16.8.2.4.1 General

This subclause (16.8.2.4) describes the interpretations of logic values occurring as literals (or in literals) after a synthesis tool has replaced named constants by their corresponding values.

16.8.2.4.2 Interpretation of the forcing and weak values ('0', '1', 'L', 'H', FALSE, TRUE)

A synthesis tool shall interpret the following values as representing a logic value 0:

- The BIT value '0'
- The BOOLEAN value FALSE
- The STD_ULOGIC values '0' and 'L'

It shall interpret the following values as representing a logic value 1:

- The BIT value '1'
- The BOOLEAN value TRUE
- The STD_ULOGIC value '1' and 'H'

This standard makes no restriction as to the interpretation of the relative strength of values.

16.8.2.4.3 Interpretation of the metalogical values ('U', 'W', 'X', '–')

16.8.2.4.4 Metalogical values in relational expressions

If the VHDL source code includes an equality operator (=) for which one operand is a static metalogical value and for which the other operand is not a static value, a synthesis tool shall interpret the equality relation as equivalent to the BOOLEAN value FALSE. If one operand of an equality relation is a one-dimensional array, and one element of that one-dimensional array is a static metalogical value, a synthesis tool shall interpret the entire equality relation as equivalent to the BOOLEAN value FALSE.

A synthesis tool shall interpret an inequality operator (/=) for which one operand is or contains a static metalogical value, and for which the other operand is not a static value, as equivalent to the BOOLEAN value TRUE.

A synthesis tool shall treat an ordering operator (<, <=, >, or >=) for which at least one operand is or contains a static metalogical value as an error.

16.8.2.4.5 Metalogical values as a choice in a case statement

If a metalogical value occurs as a choice, or as an element of a choice, in a case statement that is interpreted by a synthesis tool, the synthesis tool shall interpret the choice as one that can never occur. That is, the interpretation that is generated is not required to contain any constructs corresponding to the presence or absence of the sequence of statements associated with the choice.

Whenever a synthesis tool interprets a case statement alternative that associates multiple choices with a single sequence of statements, it shall produce an interpretation consistent with associating the sequence of statements with each choice individually.

Whenever a synthesis tool interprets a selected signal assignment statement, it shall interpret the selected signal assignment statement as if it were the case statement in the equivalent process as defined in 11.6.

16.8.2.4.6 Metalogical values in logical, arithmetic, and shift operations

When a static metalogical value occurs as all of, or one element of, an operand to a logical, arithmetic, or shift operation, and when the other operand to the operation is not a static value, a synthesis tool shall treat the operation as an error. An arithmetic operation is one of the operators +, –, *****, /, **mod**, **rem**, **abs**, and ******.

16.8.2.4.7 Metalogical values in concatenate operations

If a static metalogical value occurs as all of, or as one element of, an operand to the concatenate (&) operator, a synthesis tool shall treat it as if it had occurred as the corresponding element of the expression formed by the concatenate operation.

16.8.2.4.8 Metalogical values in type conversion and sign-extension functions

If a static metalogical value occurs as all of, or as one element of, the operand of a type conversion or sign-extension function, a synthesis tool shall treat it as if it had occurred as the corresponding element of the expression formed by the function call.

16.8.2.4.9 Metalogical values used in assignment references

A synthesis tool shall accept a static metalogical value used as all of, or as one element of, a value expression in an assignment statement, but is not required to provide any particular interpretation of that metalogical value.

16.8.2.4.10 Interpretation of the high-impedance value ('Z')

If the static value 'Z' occurs as a value expression in a signal assignment statement, a synthesis tool shall interpret the assignment as implying the equivalent of a three-state buffer that is disabled when the conditions under which the assignment occurs is true. The output of the three-state buffer is the target of the assignment. The input of the three-state buffer is the logic network that represents the value of the target apart from any assignments to 'Z'.

If the 'Z' occurs as one or more elements of a value expression in a signal assignment statement, a synthesis tool shall interpret each such occurrence as implying the equivalent of a three-state buffer in the manner defined by the preceding paragraph.

This standard does not specify an interpretation when a static value 'Z' occurs as all of, or one bit of, a value expression in a variable assignment statement.

Whenever a static high-impedance value occurs in any context other than a value expression in an assignment statement, a synthesis tool shall treat it as equivalent to a static metalogical value.

NOTE—A signal assignment statement that assigns one or more bits of a signal to 'Z' unconditionally implies the equivalent of a three-state buffer that is always disabled. A synthesis tool may choose to ignore such assignments.

16.8.3 The STD_MATCH function and predefined matching relational operators

The NUMERIC_STD package defines functions named STD_MATCH that, like the predefined matching relational operators, provide wild card matching for the don't care value. Whenever the STD_MATCH function compares two actual parameters that are STD_ULOGIC values, it returns TRUE if and only if:

- Both values are neither metalogical or high-impedance values and the values are the same, or
- One value is '0' and the other is 'L', or
- One value is '1' and the other is 'H', or
- At least one of the values is the don't care value ('–').

Whenever the STD_MATCH function compares two actual parameters that are one-dimensional arrays whose elements belong to the STD_ULOGIC type or to one of its subtypes, it returns TRUE if and only if:

a) The operands have the same length, and
b) STD_MATCH applied to each pair of matching elements returns TRUE.

When one of the actual parameters to the STD_MATCH function or a predefined matching equality operator is a static value and the other is not, a synthesis tool shall interpret the call to the STD_MATCH function or predefined matching equality operator as equivalent to an equality test on matching elements of the actual parameters, excepting those elements of the static value that are equal to '–'.

When one of the operands of a predefined matching inequality operator is a static value and the other is not, a synthesis tool shall interpret the call to the predefined matching inequality operator as equivalent to a call to the predefined matching equality operator followed by application of the **not** operator to the result.

NOTE—If any actual parameter passed to STD_MATCH is or contains a metalogical or high-impedance value other than '–', the function returns FALSE.

16.8.4 Signal edge detection

Wherever a synthesis tool interprets a particular expression as the edge of a signal, it shall also interpret the function RISING_EDGE as representing a rising edge and the function FALLING_EDGE as representing a falling edge, where RISING_EDGE and FALLING_EDGE are the functions declared either by the package STD_LOGIC_1164 or by the package NUMERIC_BIT.

16.8.5 Packages for arithmetic using bit and standard logic values

16.8.5.1 General

Four VHDL packages for arithmetic using bit and standard logic values are defined by this standard. The NUMERIC_BIT and NUMERIC_BIT_UNSIGNED packages are based on the VHDL type BIT, while the NUMERIC_STD and NUMERIC_STD_UNSIGNED packages are based on the type STD_ULOGIC.

Simulations based on the subprograms of the NUMERIC_BIT and NUMERIC_BIT_UNSIGNED packages ordinarily require less execution time, because the subprograms do not have to deal with operands containing metalogical or high-impedance values. Use of the subprograms of the NUMERIC_STD and NUMERIC_STD_UNSIGNED packages allow simulation to detect the propagation or generation of metalogical values.

The NUMERIC_BIT package defines a one-dimensional array type named SIGNED and a one-dimensional array type named UNSIGNED. The type UNSIGNED represents an unsigned binary integer with the most significant bit on the left, while the type SIGNED represents a two's-complement binary integer with the most significant bit on the left. In particular, a one-element SIGNED one-dimensional array represents the integer values –1 and 0.

The NUMERIC_STD package defines a one-dimensional array type named UNRESOLVED_SIGNED and a one-dimensional array type named UNRESOLVED_UNSIGNED, and aliases U_SIGNED and U_UNSIGNED for these two types, respectively. The package also defines a subtype named SIGNED of the base type UNRESOLVED_SIGNED and a subtype named UNSIGNED of the base type UNRESOLVED_UNSIGNED. Whereas the base types have unresolved elements, the subtypes associate the resolution function RESOLVED from the STD_LOGIC_1164 package with the elements. UNRESOLVED_UNSIGNED and UNSIGNED represent unsigned binary integers, and UNRESOLVED_SIGNED and SIGNED represent two's-complement binary integers, in the same way as the types UNSIGNED and SIGNED, respectively, from the NUMERIC_BIT package.

The NUMERIC_BIT_UNSIGNED package provides the same operations as those provided by the NUMERIC_BIT package on UNSIGNED operands, but operating on BIT_VECTOR operands interpreted as representing unsigned binary integers. Similarly, the NUMERIC_STD_UNSIGNED package provides the same operations as those provided by the NUMERIC_STD package on UNSIGNED operands, but operating on STD_ULOGIC_VECTOR operands interpreted as representing unsigned binary integers.

The four packages are mutually incompatible, and only one shall be used in any given design unit. To facilitate changing from one package to the other, most of the subprograms declared in one package are also declared for corresponding parameters in the other. Exceptions are when:

a) The NUMERIC_BIT package declares the functions RISING_EDGE and FALLING_EDGE; the corresponding functions for STD_ULOGIC are declared by the STD_LOGIC_1164 package.

b) The NUMERIC_STD package declares the STD_MATCH functions, which give special treatment to the don't care value, whereas the BIT-based types of the NUMERIC_BIT package have no don't care values.

c) The NUMERIC_STD package declares the TO_01, TO_X01, TO_X01Z, TO_UX01, and IS_X functions, which may be applied to SIGNED and UNSIGNED values, and which map the element values to the STD_ULOGIC values '0', '1', and metalogical and high-impedance values.

If a null array is furnished as an actual parameter to any subprogram declared by the packages, a synthesis tool shall treat it as an error.

All one-dimensional array return values that are not null array values are normalized so that the direction of the index range is **downto** and the right bound is 0. A one-dimensional array return value that is a null array has the index range 0 **downto** 1.

All of the packages defined in this subclause (16.8) shall be analyzed into the library symbolically named IEEE.

16.8.5.2 Allowable modifications

Vendors of tools conforming to this standard shall not modify the package declarations. However, a vendor may provide package bodies for any of the packages in which subprograms are rewritten for more efficient simulation or synthesis, provided that the behavior of the rewritten subprograms remains the same under simulation. The behavior of the original and rewritten subprograms are the same if, for any combination of input values, they return the same return values. The text of messages associated with assertions may differ in the rewritten subprogram.

The package bodies for the NUMERIC_BIT and NUMERIC_STD packages declare a constant named NO_WARNING that has the value FALSE. A user may set NO_WARNING to TRUE and reanalyze the package body to suppress warning messages generated by calls to the functions in these packages. For this reason:

— A tool vendor who rewrites the package body shall preserve the declaration of the NO_WARNING constant to allow a user to suppress warnings by editing and reanalyzing the package body.

— A simulation tool vendor who provides a preanalyzed version of the package body should also provide a mechanism for suppressing warning messages generated by the package functions.

16.8.5.3 Compatibility with previous editions of IEEE Std 1076

The following functions from the packages are compatible with IEEE Std 1076-1993 and subsequent editions of this standard but not with a previous edition, IEEE Std 1076-1987:

— binary "xnor"

— "sll"

— "srl"

— "rol"

— "ror"

— "sla"

— "sra"

To use these functions with a VHDL-based system that has not yet been upgraded to be compatible with IEEE Std 1076-1993 and subsequent editions, a user or vendor may comment out the subprogram declarations and subprogram bodies.

The following functions from the packages are compatible with this standard but not with previous editions:

— unary "and"

— unary "nand"

— unary "or"

— unary "nor"

— unary "xor"

— unary "xnor"

To use these functions with a VHDL-based system that has not yet been upgraded to be compatible with this edition of this standard, a user or vendor may comment out the subprogram declarations and subprogram bodies.

In addition, IEEE Std 1076-1993 and subsequent editions support a character set that includes the copyright symbol (©). However, IEEE Std 1076-1987 does not support this same character set. Therefore, in order to use the packages with a system that has not yet been upgraded to be compatible with IEEE Std 1076-1993

and subsequent editions, a user or vendor may replace the copyright symbol within the sources of those packages by a left parenthesis, a lowercase "c," and a right parenthesis.

16.8.5.4 The package texts

The texts of the packages (both package declarations and package bodies) are included with this standard (see Annex A). Those texts are an official part of this standard.

16.9 Standard synthesis context declarations

The library denoted by the library logical name IEEE contains context declarations IEEE_BIT_CONTEXT and IEEE_STD_CONTEXT.

```
context IEEE_BIT_CONTEXT is
   library IEEE;
   use IEEE.NUMERIC_BIT.all;
end context IEEE_BIT_CONTEXT;

context IEEE_STD_CONTEXT is
   library IEEE;
   use IEEE.STD_LOGIC_1164.all;
   use IEEE.NUMERIC_STD.all;
end context IEEE_STD_CONTEXT;
```

16.10 Fixed-point package

The library denoted by the library logical name IEEE contains packages FIXED_FLOAT_TYPES, FIXED_GENERIC_PKG, and FIXED_PKG.[12] The following conformance rules shall apply as they pertain to the use and implementation of these packages:

a) The package declarations may be modified to include additional data required by tools, but modifications shall in no way change the external interfaces or simulation behavior of the description. It is permissible to add comments and/or attributes to the package declarations, but not to change or delete any original lines of the approved package declaration.

b) The FIXED_GENERIC_PKG package body and the FIXED_PKG package instantiation declaration provided in the machine-readable files accompanying this standard (see Annex A) represent the formal semantics of the implementation of the FIXED_GENERIC_PKG and FIXED_PKG packages. Implementers of these packages may choose to simply compile the package body and package instantiation declaration as it is, or they may choose to implement the packages in the most efficient form available to the user. Implementers shall not implement semantics that differ from the formal semantics provided herein.

The text of the FIXED_GENERIC_PKG package (both package declaration and package body) and the text of the FIXED_PKG instantiated package are included with this standard (see Annex A). Those texts are official parts of this standard.

[12]The packages FIXED_GENERIC PKG, FIXED_PKG, and FIXED_FLOAT_TYPES were modified and used with permission from Eastman Kodak Company © 2006.

16.11 Floating-point package

The library denoted by the library logical name IEEE contains packages FLOAT_GENERIC_PKG and FLOAT_PKG.[13] The following conformance rules shall apply as they pertain to the use and implementation of these packages:

a) The package declarations may be modified to include additional data required by tools, but modifications shall in no way change the external interfaces or simulation behavior of the description. It is permissible to add comments and/or attributes to the package declarations, but not to change or delete any original lines of the approved package declaration.

b) The FLOAT_GENERIC_PKG package body and the FLOAT_PKG package instantiation declaration provided in the machine-readable files accompanying this standard (see Annex A) represent the formal semantics of the implementation of the FLOAT_GENERIC_PKG and FLOAT_PKG packages. Implementers of these packages may choose to simply compile the package body and package instantiation declaration as it is, or they may choose to implement the packages in the most efficient form available to the user. Implementers shall not implement semantics that differ from the formal semantics provided herein.

The text of the FLOAT_GENERIC_PKG package (both package declaration and package body) and the text of the FLOAT_PKG instantiated package are included with this standard (see Annex A). Those texts are official parts of this standard.

[13]The packages FLOAT_GENERIC_PKG and FLOAT_PKG were modified and used with permission from Eastman Kodak Company © 2006.

17. VHDL Procedural Interface overview

17.1 General

The VHDL Procedural Interface (VHPI) is an application-programming interface to VHDL tools that allows programmatic access to a VHDL model during its analysis, elaboration, and execution. The VHPI is described in this clause, subsequent clauses through to Clause 23, and Annex B.

17.2 Organization of the interface

17.2.1 General

The VHPI consists of two aspects:

— An *information model* that represents the topology and state of a VHDL model

— A number of functions that operate on the information model to access or affect the state of the VHDL model and that interact with tools during analysis, elaboration, or execution of the VHDL model

A *tool* is a program that maintains a representation of a VHDL model and provides the VHPI functions. A *VHPI program* is a program that calls the VHPI functions.

The information model is expressed in an object-oriented manner as a set of *classes* that bear *relationships* to one another. The classes are data types that have data *properties* and subprogram *operations*. A *subclass* may be derived from one or more *superclasses*, in which case it *inherits* the properties and operations of its superclasses. An *object* is an instance of a class and of any superclasses of that class. The *most specialized* class of an object is the class of which the object is a member and that has no subclass of which the object is also a member. An *abstract* class cannot be the most specialized class of any object; however, it may be a superclass of a non-abstract class that is the most specialized class of an object.

Some objects are *static*; that is, once created, they remain in existence until termination of the tool. Other objects are *dynamic*; that is, once created, they may cease to exist at a later time during execution of the tool.

The properties of a class represent data that is characteristic of an object of the class. The VHPI provides functions that allow a VHPI program to access and modify the values of properties of a given object. By using such functions, a VHPI program can access and modify values of VHDL objects in a VHDL model.

In addition to the inheritance relationship, a class may bear an association relationship with one or more other classes. A *one-to-one* association means that an object of a class is associated with at most one object of the second class. A *one-to-many* association means that an object of the class is associated with possibly more than one object of the second class. The VHPI provides functions that allow a VHPI program to traverse associations; that is, to locate objects that are associated with a given object.

The information model contains two sub-models. The first, referred to as the *library information model*, represents the design units that comprise a VHDL model after analysis and prior to elaboration. The second, referred to as the *design hierarchy information model*, represents the elaborated VHDL model. It contains instances, created through elaboration, of objects from the library information model. The design hierarchy information model may be used by a tool that simulates the VHDL model to gain access to the state of the VHDL model during execution. The design hierarchy information model includes associations with objects in the library information model, allowing navigation between the information models. The information model also contains objects representing the tool and its environment.

A VHPI program can interact with a tool by providing *callback* functions. Such functions are identified to the tool using VHPI registration functions. The tool then calls the functions in response to events specified during registration. Such events include phases of tool execution, phases of model simulation, and changes of value of VHDL objects.

The VHPI also provides *utility* functions for such purposes as printing, error checking, and tool control.

In this standard, the VHPI information model is described using the UML notation. UML is described in ISO/IEC 19501:2005. The VHPI is ISO C-compliant. The VHPI functions are expressed as C functions, and the data and arguments used by the functions are expressed as C data types. ISO C is described in ISO/IEC 9899:1999, as corrected by ISO/IEC 9899:1999/Cor 1:2001 and ISO/IEC 9899:1999/Cor 2:2004.

17.2.2 VHPI naming conventions

Named items in the VHPI conform to the following conventions:

- The names of functions consist of the letters vhpi followed by an underline character and one or more words, each of which consists of lowercase letters, with a single underline character between words.
- The names of items other than functions consist of the letters vhpi followed by one or more words, each of which consists of an uppercase letter followed by zero or more lowercase letters, with no character between words.
- The names of types end in an uppercase letter T.
- The names of enumeration constants that correspond to classes end in an uppercase letter K.
- The names of enumeration constants that correspond to properties end in an uppercase letter P.
- The names of enumeration constants that correspond to one-to-many associations end in a lowercase letter s, indicating plurality.
- Some words are abbreviated, for example, decl for declaration, stmt for statement, conc for concurrent, seq for sequential, and subp for subprogram.

In this standard, C identifiers are formatted in a monospaced font to enhance readability of the text.

17.3 Capability sets

The VHPI is divided into a number of capability sets, each of which provides a subset of the VHPI operations, properties, and functions. Corresponding to each capability set, there is an enumeration constant of type `vhpiCapabilitiesT` defined in the VHPI header file (see Annex B).

The VHPI capability set names and corresponding enumeration constants are:

- Hierarchy set: `vhpiProvidesHierarchy`.

  A tool that implements this hierarchy set shall provide access to objects in the design hierarchy information model that represent statically elaborated regions and declarations and shall provide access to the values of declared objects.
- Static access set: `vhpiProvidesStaticAccess`. This set requires the hierarchy set.

  A tool that implements the static access set shall additionally provide access to objects in the design hierarchy information model that represent statically elaborated statements and the expressions within them.
- Connectivity set: `vhpiProvidesConnectivity`. This set requires the hierarchy set.

  A tool that implements the connectivity set shall additionally provide access to objects in the design hierarchy information model that represent drivers, contributors, loads, and port associations.
- Post-analysis set: `vhpiProvidesPostAnalysis`.

A tool that implements the post-analysis set shall provide access to objects in the library information model and shall provide access to the values of declared objects that are initialized with locally static expressions.

— Basic foreign model set: `vhpiProvidesForeignModel`. This set requires the hierarchy set.

  A tool that implements the basic foreign model set shall additionally support creation of foreign models and foreign model callbacks and shall provide access to objects in the design hierarchy information model that represent foreign models.

— Advanced foreign model set: `vhpiProvidesAdvancedForeignModel`. This set requires the basic foreign model set.

  A tool that implements the advanced foreign model set shall additionally support creation of foreign drivers and processes and scheduling of transactions on foreign drivers.

— Save/restart set: `vhpiProvidesSaveRestart`.

  A tool that implements the save/restart set shall support save and restart of foreign models, use of the `vhpi_put_data` and `vhpi_get_data` functions, save and restart callbacks, and shall provide access to the `Id` and `SaveRestartLocation` properties.

— Reset set: `vhpiProvidesReset`.

  A tool that implements the reset set shall support reset of foreign models and reset callbacks.

— Basic debug and runtime simulation set: `vhpiProvidesDebugRuntime`. This set requires the static access set and the connectivity set.

  A tool that implements the debug and runtime simulation set shall support use of the `vhpi_control`, `vhpi_get_time`, and `vhpi_get_next_time` functions; object value change callbacks for signals, ports, and drivers; time and action callbacks; and updating of signals, ports, and drivers.

— Advanced debug and runtime simulation set: `vhpiProvidesAdvancedDebugRuntime`. This set requires the basic debug and runtime simulation set.

  A tool that implements the advanced debug and runtime simulation set shall additionally support object value change callbacks for variables, updating of variables, and the `LineOffset` property.

— Dynamic elaboration set: `vhpiProvidesDynamicElab`. This set requires the debug and runtime simulation set.

  A tool that implements the dynamic elaboration set shall additionally provide access to objects that represent dynamically elaborated regions, declarations, and constructs.

If a tool specifies that it implements a given capability set, it shall provide all of the operations, properties, and functions specified for the capability set. If the capability set requires one or more other capability sets, the tool shall also implement the required capability sets. A tool shall provide a value for the `vhpiCapabilitiesP` property of the `tool` class that specifies the capability sets that the tool implements.

If a VHPI program calls an operation or function that is not provided in the capability sets provided by a tool, the function shall raise a VHPI error condition. Similarly, if a VHPI program accesses a property that is not provided in the capability sets provided by a tool, the access function shall raise a VHPI error condition. In both cases, the error message returned by a subsequent call to `vhpi_check_error` shall indicate that the operation is not implemented.

NOTE—A minimal implementation of the VHPI need only provide the function interface described in Clause 23 and Annex B, with none of the capability sets described in this subclause being implemented by the tool. In such a minimal implementation, calls to functions would, in most cases, raise a VHPI error condition.

17.4 Handles

17.4.1 General

A *handle* is an opaque reference to an object in the VHPI information model. It is represented as a value of the data type `vhpiHandleT` (see Annex B); however, the interpretation of the representation is implementation defined. A handle allows a VHPI program to refer to an object without assuming details of the representation of the object. The VHPI provides functions that operate on objects referred to by handles. The particular operations that are legal for an object referred to by a handle depend on the class of the object. The class is identified by the `Kind` property of the object.

In this standard, if an object is described as being of a given class, the object may be of the given class, provided the class is not an abstract class or any non-abstract subclass of the given class.

NOTE—The `Kind` property of an object identifies the most specific class of the object, that is, the class for which no subclass is also a class of the object.

17.4.2 Handle creation

A handle is created by a tool as the result of one of the following functions called by a VHPI program:

- `vhpi_handle_by_name`, which returns a handle that refers to an object identified by a name
- `vhpi_handle_by_index`, which returns a handle that refers to an object in an ordered one-to-many association
- `vhpi_handle`, which returns a handle that refers to the object in a one-to-one association
- `vhpi_create`, which creates or modifies an object, such as a driver, a process statement, or a collection, and returns a handle that refers to the object
- `vhpi_register_cb`, which returns a handle that refers to the callback object
- `vhpi_register_foreignf`, which returns a handle that refers to the callback object
- `vhpi_iterator`, which returns a handle that refers to an iterator
- `vhpi_scan`, which returns a handle that refers to an object referenced by an iterator

A tool shall support multiple VHPI programs, each of which acquires handles. The way in which a tool implements handles shall allow a VHPI program to function correctly independently of other VHPI programs executing concurrently. A tool may share between VHPI programs resources associated with the implementation of handles and the objects to which they refer. However, the occurrence of such sharing shall not alter the effect of the VHPI programs.

If a tool creates two handles that refer to the same object, the tool may create two distinct handles or may provide the same handle in both cases. Two distinct handles that refer to the same object are equivalent.

NOTE—The number of handles that an implementation can create may be constrained by the capacity of the host system.

17.4.3 Handle release

The function `vhpi_release_handle` called by a VHPI program causes a tool to release a handle. If a tool shares resources associated with handles and one VHPI program releases a handle, other VHPI programs shall be able to continue to refer to objects using handles that they have not released.

The tool may reclaim resources associated with the representation of a released handle.

NOTE 1—It is recommended that a VHPI program release handles when they are no longer needed.

NOTE 2—A tool may reclaim resources associated with a handle when the handle is released by a VHPI program, provided the requirements of 17.4 are met. As a consequence, resources might not be reclaimed immediately upon release of a handle by a VHPI program, as the resources may be associated with handles in use by other VHPI programs.

17.4.4 Handle comparison

The function `vhpi_compare_handles` compares handles. It returns the value `vhpiTrue` if the handles are equivalent (that is, they refer to the same object); otherwise it returns the value `vhpiFalse`.

17.4.5 Validity of handles

The *lifetime* of an object is the duration of existence of the object in the VHPI information model. A static object is created at some time during the execution of a tool and exists until termination of the tool. A dynamic object is created at some time during the execution of a tool and may cease to exist at a later time during the execution of the tool, either as a consequence of execution of the VHDL model or of removal by a VHPI program.

A tool can create a handle that refers to an object only during the lifetime of the object. A handle is said to be *valid* from the time of its creation until the time at which it is released, or until the object that it refers to ceases to exist, or until termination of the tool; at other times it is *invalid*. A VHPI program that attempts to refer to an object using an invalid handle is erroneous.

NOTE—A VHPI program that attempts to release an invalid handle is also erroneous.

18. VHPI access functions

18.1 General

This clause describes the VHPI functions that are used by VHPI programs to access the information model of a VHDL model.

18.2 Information access functions

18.2.1 General

The VHPI information access functions allow a VHPI program to navigate an association between objects.

The VHPI header file defines enumeration types that contain enumeration constants corresponding to association roles specified implicitly or explicitly in the information model. The name of each enumeration constant is the name of the corresponding role prefixed with the letters `vhpi`.

18.2.2 One-to-one association traversal

The VHPI header file defines the enumeration type `vhpiOneToOneT` that contains enumeration constants corresponding to one-to-one association roles.

If the information model includes a one-to-one association that is navigable from a reference class to a target class, the function `vhpi_handle` navigates from an object of the reference class to an object of the target class (see 23.20).

Examples:

Given the information model described by the UML class diagram shown in Figure 1

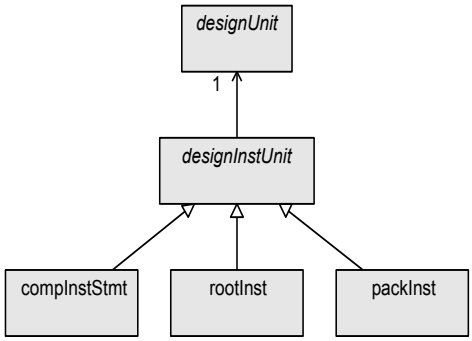

Figure 1—UML class diagram

the following VHPI program navigates from an object of the `compInstStmt` class to an object of the `designUnit` class using the enumeration constant `vhpiDesignUnit`.

```
void get_binding_info(vhpiHandleT instHdl) {
  char duName[MAXSTR];
  char libName[MAXSTR];
  vhpiHandleT duHdl;
```

```
  switch (vhpi_get(vhpiKindP, instHdl)) {
  case vhpiCompInstStmtK:
  case vhpiRootInstK:
  case vhpiPackInstK:
    duHdl = vhpi_handle(vhpiDesignUnit, instHdl);
    sprintf (duName, "%s", vhpi_get_str(vhpiUnitNameP, duHdl));
    sprintf(libName, "%s", vhpi_get_str(vhpiLibLogicalNameP, duHdl));
    vhpi_printf("design unit name %s in library %s\n", duName, libName);
    break;
  default:
    break;
  }/* end switch */
}/* get_binding_info() */
```

Given the information model described by the UML class diagram shown in Figure 2

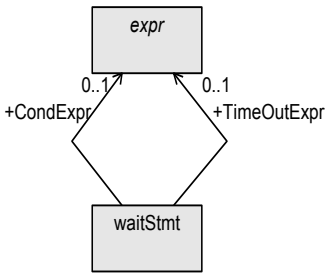

Figure 2—UML class diagram

the following VHPI program navigates from an object of the `waitStmt` class to one object of the `expr` class using the enumeration constant `vhpiCondExpr` and to a second object of the `expr` class using the enumeration constant `vhpiTimeOutExpr`.

```
vhpiHandleT stmtHdl, condHdl, timeHdl;
if (vhpi_get(vhpiKindP, stmtHdl) == vhpiWaitStmtK) {
  condHdl = vhpi_handle(vhpiCondExpr, stmtHdl);
  timeHdl = vhpi_handle(vhpiTimeOutExpr, stmtHdl);
}
```

18.2.3 One-to-many association traversal

The VHPI header file defines the enumeration type `vhpiOneToManyT` that contains enumeration constants corresponding to one-to-many association roles.

If the information model includes a one-to-many association that is navigable from a reference class to a target class, the function `vhpi_iterator` navigates from an object of the reference class to a set of objects of the target class (see 23.24).

If the information model includes an ordered one-to-many association that is navigable from a reference class to a target class, the function `vhpi_handle_by_index` navigates from an object of the reference class to an object of the target class (see 23.21).

NOTE 1—A VHPI program can use the `vhpi_scan` function to access the objects referred to by an iterator.

NOTE 2—If the association navigated by the `vhpi_iterator` function is not an ordered association, the order of objects returned by applying `vhpi_scan` to the iterator is not defined.

Example:

```
vhpiHandleT instHdl, instIter;
/* get all sub-instances of a scope instance */
instIter = vhpi_iterator(vhpiInternalRegions, instHdl);
if (instIter) {
  while (instHdl = vhpi_scan(instIter)) {
    vhpi_printf("found instance %s\n",
                vhpi_get_str(vhpiNameP, instHdl));
  }
}
```

18.3 Property access functions

18.3.1 General

The VHPI property access functions allow a VHPI program to access property values of objects.

The VHPI header file defines enumeration types that contain enumeration constants corresponding to properties of classes specified in the information model. The name of each enumeration constant is the name of the corresponding property prefixed with the letters `vhpi` and suffixed with the uppercase letter `P`.

18.3.2 Integer and Boolean property access function

The VHPI header file defines the enumeration type `vhpiIntPropertyT` that contains enumeration constants corresponding to integer and Boolean properties. The header file defines the type `vhpiIntT` that is used to represent values of integer and Boolean properties. The header file defines the integer constant `vhpiFalse` that is used to represent the value of a Boolean property that is false and the integer constant `vhpiTrue` that is used to represent the value of a Boolean property that is true.

The function `vhpi_get` accesses an integer or Boolean property of an object (see 23.10).

NOTE—Some properties may legally take on the same value as the constant `vhpiUndefined`. In such cases, a VHPI program should use the `vhpi_check_error` to determine whether a call to `vhpi_get` resulted in an error.

18.3.3 String property access function

The VHPI header file defines the enumeration type `vhpiStrPropertyT` that contains enumeration constants corresponding to string properties.

The function `vhpi_get_str` accesses a string property of an object (see 23.17).

NOTE 1—Successive calls to `vhpi_get_str` may use the same storage for the results. A VHPI program that needs to save the result of a call to `vhpi_get_str` should copy the result before subsequent calls to the function. (See Clause 23.)

NOTE 2—String properties that represent VHDL pathnames and extended identifiers may contain non-letter graphic characters, such as '\'. VHPI programs that use C string library functions or printf functions to operate on such strings should ensure that the special characters are not interpreted as escape characters by the functions.

18.3.4 Real property access function

The VHPI header file defines the enumeration type `vhpiRealPropertyT` that contains enumeration constants corresponding to real properties. The header file defines the type `vhpiRealT` that is used to represent values of real properties.

The function `vhpi_get_real` accesses a real property of an object (see 23.16).

NOTE—A VHPI program should use the `vhpi_check_error` to determine whether a call to `vhpi_get_real` resulted in an error.

18.3.5 Physical property access function

The VHPI header file defines the enumeration type `vhpiPhysPropertyT` that contains enumeration constants corresponding to physical properties. The header file defines the struct type `vhpiPhysT` that is used to represent values of physical properties. The member high of the struct type represents the most significant 32 bits of the position number of a value, and the member low represents the least significant 32 bits of the position number of the value.

The function `vhpi_get_phys` accesses a physical property of an object (see 23.15).

NOTE—A VHPI program should use the `vhpi_check_error` to determine whether a call to `vhpi_get_phys` resulted in an error.

18.4 Access by name function

If a class in the information model has the `vhpiFullNameP` property (see 19.4.7), the function `vhpi_handle_by_name` (see 23.22) navigates to an object of the class.

19. VHPI information model

19.1 General

This clause describes the VHPI information model using the Unified Modeling Language (UML) (ISO/IEC 19501:2005). The clause specifies the classes, subclass relationships, associations, properties, and operations of the information model. Part of this clause is included here in textual form. The remainder of this clause is included in machine-readable form, comprising a navigable representation of the information model.

The information model described here allows representation of VHDL models that conform to IEEE Std 1076-2002. Certain aspects of the language added in the current revision of this standard cannot be represented by the information model. It is expected that a subsequent revision of this standard will extend the information model to allow representation of those aspects.

19.2 Formal notation

19.2.1 General

The information model is described using a set of UML class diagrams. The diagrams specify the classes that are included in the information model, the subclass relationships that exist between classes, the properties and operations of classes, and the associations that exist between objects of classes.

Each association is annotated with the *navigability* of the association. If the association is navigable from an object of one class to an object of a second class, the first class is said to be the *reference class*, and the second class is said to be the *target class*. The object of the reference class is said to be the *reference object*, and an object of the target class is said to be a *target object*. An association may be navigable in one direction only (in which case, it is shown with an arrow indicating the direction of navigability) or it may be navigable in both directions (in which case it is shown with no arrow).

Each association is annotated with the *multiplicity* of the association in the direction of navigation of the association. One-to-one associations are those that have a multiplicity of 1 or 0..1 in the direction of navigation. One-to-many associations are those that have a multiplicity of 0..* or 1..* in the direction of navigation.

Some associations are annotated with a *role name* in the direction of navigation of the association. If an association is not so annotated, the role name is implicitly the name of the target class.

Some one-to-many associations are annotated with the *ordered* constraint. The description of the association includes a specification of the order of occurrence of target objects within the association.

In certain cases, a class inherits a given property or association from more than one superclass, or has a given property or association and also inherits the property or association from a superclass. In such cases, the class does not replicate the property or association. Rather, the class has a single occurrence of the property or association. The meaning of the property or association is the same for all classes in which it is specified.

19.2.2 Machine-readable information model

The machine-readable form of the information model is part of this clause. The following aspects of the machine-readable form are normative:

a) The partitioning of class diagrams, class specification, and association specifications into packages

b) The class diagrams

c) For each package specification:
   — The package name
   — The text of the documentation
   — The list of classes provided by the package

d) For each class specification:
   — The text of the documentation
   — The specification of whether the class is abstract
   — The cardinality
   — The name and signature of operations
   — The name, supplying class, and type of properties (referred to as “attributes” in the machine-readable form)
   — The role names and target classes of associations
   — The specialized class and supplier class of generalization relationships

e) For each property specification:
   — The name, type, and supplying class of the property
   — The text of the documentation

f) For each operation specification:
   — The name, signature, and supplying class of operations
   — The text of the documentation

g) For each association specification:
   — If an association is navigable in a given direction, the target class role name, and the target class
   — Otherwise, the role name is shown as “Not Named”

h) For each navigable association role specification:
   — The role name and target class
   — The text of the documentation
   — The multiplicity (referred to as the “cardinality” in the machine-readable form)
   — The navigability

NOTE—Other aspects of the machine-readable form do not form part of this standard. They occur as a side effect of the software program used to develop this standard.

19.3 Class inheritance hierarchy

The UML description of the VHPI information model is partitioned into several UML packages. Each package defines one or more classes and includes one or more class diagrams. The class diagrams of all of the packages jointly specify the inheritance hierarchy of the UML description, that is, the set of inheritance relationships that exist between all of the classes of the information model.

The class `base` forms the root of the inheritance hierarchy; all other classes inherit directly or indirectly from it. A single virtual object of class `null` represents the context in which the VHPI tool executes and is accessed using a `NULL` handle. Other classes represent aspects of the VHDL model and VHPI programs being processed by the VHPI tool.

For each class, this clause and the documentation in the machine-readable form of the information model jointly describe the properties, operations, and associations defined in the information model. The class also inherits properties, operations, and associations defined for its superclasses.

19.4 Name properties

19.4.1 General

This subclause (19.4) describes certain properties of objects that relate to the names of VHDL named entities or constructs. Other name-related properties are described in the documentation in the machine-readable form of the information model.

19.4.2 Implicit labels of statements

19.4.2.1 General

Certain properties that relate to names derive their values from labels of statements. In cases where the label of such a statement is optional, this subclause (19.4.2) describes rules for determining an implicit label that is used in the value of the property.

19.4.2.2 Implicit labels of loop statements

For each loop statement that occurs immediately within a given declarative region, there corresponds a unique sequence number, determined as follows. The loop statements are ordered according to the order of occurrence of their first lexical elements in the text of the declarative region. The sequence number of the first loop statement in the ordering, if any, is 0. The sequence number of each subsequent loop statement in the ordering, if any, is one greater than that of the preceding loop statement.

If a loop statement is unlabeled, an implicit label is defined for use in name properties. The implicit label is a sequence of characters starting with an underline character, followed by the letter 'L' or 'l', further followed by the loop sequence number of the loop statement expressed in decimal without leading insignificant zero digits. The choice between the letter 'L' and 'l' is implementation defined.

Example:

In the following VHDL procedure body, the implicit loop labels are indicated in comments.

```
procedure LOOP_EXAMPLE is
begin
   loop  -- _L0
      L: for I in 1 to 10 loop   -- explicitly labeled,
                                 -- so no implicit label defined
         while TEST loop  -- _L2
            ...
         end loop;
      end loop L;
   end loop;
   loop   -- _L3
         ...
   end loop;
end procedure LOOP_EXAMPLE;
```

19.4.2.3 Implicit labels of concurrent statements

For each concurrent statement that is a process statement or is equivalent to a process statement and that occurs immediately within a given declarative region, there corresponds a unique sequence number, determined as follows. The statements are ordered according to their order of occurrence in the text of the declarative region. In the case of statements occurring immediately within an entity declaration, a block

statement, or a generate statement, the sequence number of the first statement in the ordering, if any, is 0. In the case of statements occurring immediately within an architecture body, the sequence number of the first statement in the ordering, if any, is one greater than that of the last statement in the ordering of the entity declaration to which the architecture body corresponds. The sequence number of each subsequent statement in the ordering of the given declarative region, if any, is one greater than that of the preceding statement.

If a concurrent statement that is a process statement or is equivalent to a process statement is unlabeled, an implicit label is defined for use in name properties. The implicit label is a sequence of characters starting with an underline character, followed by the letter 'P' or 'p', further followed by the sequence number of the statement expressed in decimal without leading insignificant zero digits. The choice between the letter 'P' and 'p' is implementation defined.

Example:

In the following VHDL model, the implicit labels are indicated in comments.

```
entity E is
   generic (G: INTEGER);
   port (S: out INTEGER);
   assert G > 0;  -- _P0
end entity E;

architecture A of E is
begin
   process is  -- _P1
   begin
      ...
   end process;
   A1: assert G > 2;
   B: block is
   begin
      WORK.PKG.PROC(G);  -- _P0
   end block B;
   S <= G;  -- _P2
end architecture A;
```

19.4.3 The Name and CaseName properties

Certain objects in the information model have both the `Name` and `CaseName` properties. If the value of the `Name` property of an object is the simple name of a named entity and the simple name is in the form of an extended identifier, the case of letters occurring in the value of the `Name` property is the same as the case of letters occurring in the extended identifier. Otherwise, the case of letters occurring in the value of the `Name` property is not specified by this standard.

For an object of class `decl` that does not represent the declaration of an anonymous named entity, the values of the `Name` and `CaseName` properties are the simple name or operator symbol of the declaration represented by the object. In determining the case of letters in the `CaseName` property, there are four cases:

— If the object represents a type declaration, either there is both an incomplete type declaration and a full type declaration, in which case the case of letters in the value of the `CaseName` property is the same as the case of letters in the identifier of the incomplete type declaration; or there is only a full type declaration, in which case the case of letters in the value of the `CaseName` property is the same as the case of letters in the identifier of the full type declaration.

— If the object represents an interface object of a subprogram, either there is both a subprogram declaration and a subprogram body, in which case the case of letters in the value of the `CaseName` property is the same as the case of letters in the identifier of the interface declaration of the subprogram declaration; or there is only a subprogram body, in which case the case of letters in the value of the `CaseName` property is the same as the case of letters in the identifier of the interface declaration of the subprogram body.

— If the object represents a subprogram body for which there is a separate subprogram declaration, the case of letters in the value of the `CaseName` property is the same as the case of letters in the designator of the subprogram specification of the subprogram declaration. Otherwise, if the object represents a subprogram body for which there is no separate subprogram declaration, the case of letters in the value of the `CaseName` property is the same as the case of letters in the designator of the subprogram specification of the subprogram body.

— If the object is none of the preceding cases, the case of letters in the value of the `CaseName` property is the same as the case of letters in the identifier or operator symbol in the declaration represented by the object.

The values of the `Name` and `CaseName` properties of an object of class `decl` that represents the declaration of an anonymous named entity are not specified by this standard.

For an object of class `rootInst`, the values of the `Name` and `CaseName` properties are the simple name of the entity declaration whose instantiation is represented by the object. The case of letters in the value of the `CaseName` property is the same as the case of letters in the identifier of the entity declaration.

For an object of class `packInst`, the values of the `Name` and `CaseName` properties are the simple name of the package declaration whose elaboration is represented by the object. The case of letters in the value of the `CaseName` property is the same as the case of letters in the identifier of the package declaration.

For an object of class `protectedTypeInst`, the values of the `Name` and `CaseName` properties are the simple name of the variable whose elaboration is represented by the object. The case of letters in the value of the `CaseName` property is the same as the case of letters in the identifier of the variable declaration.

For an object of class `blockStmt`, `eqProcessStmt`, or `compInstStmt`, or for an object of class `generateStmt` other than an object of class `forGenerate` in the design hierarchy information model, or for an object of class `loopStmt`, the values of the `Name` and `CaseName` properties are the label, either explicit or implicit (see 19.4.2), of the statement represented by the object. The case of letters in the value of the `CaseName` property is the same as the case of letters in the label of the statement.

For an object of class `forGenerate` in the design hierarchy information model, the values of the `Name` and `CaseName` properties are a string of the form

*generate_statement*_label (literal)

The string includes no leading, trailing, or embedded space characters between lexical elements. The generate statement label is the label of the generate statement represented by the object, and the literal is the value of the generate parameter corresponding to the instance of the generate statement represented by the object. If the generate parameter is of an integer type, the literal is a numeric literal whose value is an integer. Otherwise, if the generate parameter is of an enumeration type, the literal is an enumeration literal whose value is of the type of the generate parameter. The case of letters in the label part of the value of the `CaseName` property is the same as the case of letters in the label of the statement. The case of letters in a numeric literal in the value of the `CaseName` property is not specified by this standard. The case of letters in an enumeration literal that is an identifier in the value of the `CaseName` property is the same as the case of letters in the identifier in the declaration of the enumeration type of which the enumeration literal is a value.

For an object of class `seqProcCall` or `funcCall` representing invocation of a subprogram other than a method of a protected type, the values of the `Name` and `CaseName` properties are the values of the `Name` and `CaseName` properties, respectively, of an object of class `subpDecl` representing the subprogram specification of the subprogram invoked.

For an object of class `seqProcCall` or `funcCall` representing invocation of a method of a protected type, the value of the `Name` property is a string of the form

*shared_variable*_name_property . *named_entity*_name_property

and the value of the `CaseName` property is a string of the form

*shared_variable*_case_name_property . *named_entity*_case_name_property

The strings include no leading, trailing, or embedded space characters between lexical elements. The shared variable name property and the shared variable case name property are the values of the `Name` and `CaseName` properties, respectively, of an object of class `decl` representing the declaration of the shared variable denoted by the prefix of the name of the subprogram invoked. The named entity name property and the named entity case name property are the values of the `Name` and `CaseName` properties, respectively, of an object of class `subpDecl` representing the subprogram specification of the subprogram invoked.

For an object of class `indexedName` representing an element of a named entity that is a declared object of an array type, the value of the `Name` property is a string of the form

*named_entity*_name_property (literal { , literal })

and the value of the `CaseName` property is a string of the form

*named_entity*_case_name_property (literal { , literal })

The strings include no leading, trailing, or embedded space characters between lexical elements. In the strings

— The named entity name property is the value of the `Name` property of the object representing the declaration of the named entity, and the named entity case name property is the value of the `CaseName` property of that object.

— Each literal is the index value of the element for the corresponding index position of the array type of the named entity. If the index subtype for a given index position is an integer type, the literal for that index position is a numeric literal whose value is an integer. Otherwise, if the index subtype for the index position is an enumeration type, the literal is an enumeration literal whose value is of the index subtype.

For an object of class `selectedName` representing an element of a named entity that is a declared object of a record type, the value of the `Name` property is a string of the form

*named_entity*_name_property . *element*_simple_name

and the value of the `CaseName` property is a string of the form

*named_entity*_case_name_property . *element*_simple_name

The strings include no leading, trailing, or embedded space characters between lexical elements. In the strings

— The named entity name property is the value of the `Name` property of the object representing the declaration of the named entity, and the named entity case name property is the value of the `CaseName` property of that object.

— The element simple name is the simple name of the element. The case of letters in the element simple name in the value of the `CaseName` property is the same as the case of letters occurring in the identifier of the element in the declaration of the record type.

For an object of class `sliceName` representing

— a slice of a named entity that is a declared object of an array type, and

— a slice in which the discrete range is in the form of a literal representing the left bound, a direction and a literal representing the right bound,

the value of the `Name` property is a string of the form

*named_entity*_name_property (literal direction literal)

and the value of the `CaseName` property is a string of the form

*named_entity*_case_name_property (literal direction literal)

The strings include no leading, trailing, or embedded space characters between lexical elements. In the strings

— The named entity name property is the value of the `Name` property of the object representing the declaration of the object denoted by the prefix of the slice, and the named entity case name property is the value of the `CaseName` property of the former object.

— The literals are the left and right bounds, respectively, of the range of the slice. If the index subtype of the object denoted by the prefix is an integer type, the literals are a numeric literals whose values are integers. Otherwise, if the index subtype of the object denoted by the prefix is an enumeration type, the literals are enumeration literals whose value is of the index subtype.

— The direction is **to** if the discrete range of the slice is an ascending range, or **downto** otherwise.

For an object of class `derefObj` in the library information model representing an element of an array variable, the values of the `Name` and `CaseName` properties are strings of the same form as the value of the `Name` and `CaseName` properties, respectively, of an object of class `indexedName`, except that:

— The named entity name property is the value of the `Name` property of an object representing the access value that designates the array variable, and the named entity case name property is the value of the `CaseName` property of that object.

— Each literal is the index value of the element for the corresponding index position of the array type of the array variable.

For an object of class `derefObj` in the library information model representing an element of a record variable, the values of the `Name` and `CaseName` properties are strings of the same form as the value of the `Name` and `CaseName` properties, respectively, of an object of class `selectedName`, except that the named entity name property is the value of the `Name` property of an object representing the access value that designates the record variable, and the named entity case name property is the value of the `CaseName` property of that object.

For an object of class `derefObj` in the library information model representing a slice of an array variable, the values of the `Name` and `CaseName` properties are strings of the same form as the value of the `Name` and `CaseName` properties, respectively, of an object of class `sliceName`, except that the named entity name property is the value of the `Name` property of an object representing the access value that designates the

array variable, and the named entity case name property is the value of the `CaseName` property of that object.

For an object of class `derefObj` in the library information model representing an entire variable, denoted by a selected name with the suffix **all**, the value of the `Name` property is a string of the form

*named_entity*_name_property . **all**

and the value of the `CaseName` property is a string of the form

*named_entity*_case_name_property . **all**

The strings include no leading, trailing, or embedded space characters between lexical elements. In the strings, the named entity name property is the value of the `Name` property of an object representing the access value that designates the variable, and the named entity case name property is the value of the `CaseName` property of that object.

It is an error if a VHPI program reads the `Name` or `CaseName` property of an object of class `derefObj` in the design hierarchy information model.

For an object of class `attrName` representing an attribute name, the value of the `Name` property is a string of the form

*named_entity*_name_property ' *attribute*_name_property

and the value of the `CaseName` property is a string of the form

*named_entity*_case_name_property ' *attribute*_case_name_property

The strings include no leading, trailing, or embedded space characters between lexical elements. In the strings

- The named entity name property is the value of the `Name` property of the object representing the prefix of the attribute name, and the named entity case name property is the value of the `CaseName` property of that object.
- For user-defined attributes, the attribute name property is the value of the `Name` property of the object representing the declaration of the attribute denoted by the attribute designator in the attribute name, and the attribute case name property is the value of the `CaseName` property of that object.
- For predefined attributes, the attribute name property and the attribute case name property are both the simple name of the attribute. The case of letters in the attribute case name property is not specified by this standard.

For an object of class `useClause` representing a reference to a declaration in a use clause, the values of the `Name` and `CaseName` properties are the values of the `Name` and `CaseName` properties, respectively, of an object of class `decl` representing the declaration.

For an object of class `designUnit` representing an analyzed design unit in the library information model, the values of the `Name` and `CaseName` properties are the simple name of the design unit. The case of letters in the simple name in the value of the `CaseName` property is the same as the case of letters occurring in the identifier of the design unit.

19.4.4 The SignatureName property

An object of class `subpDecl`, `charLiteral`, or `enumLiteral` has the `SignatureName` property. The value of the property is a string that is the signature (see 4.5.3) of the subprogram or enumeration literal, as appropriate, represented by the object. Similarly, an object of class `subpCall` has the `SignatureName` property. The value of the property is a string that is the signature of the subprogram invoked by the procedure call statement or function call represented by the object.

The signature includes a type mark for each parameter of the subprogram, and that type mark denotes the base type of the parameter. If the subprogram is a function, the signature includes the reserved word **return** and a further type mark that denotes the base type of the return type of the function. The case of letters in the value of the `SignatureName` property is not specified by this standard.

19.4.5 The UnitName property

Objects of class `designUnit` in the library information model have the `UnitName` property. The value of the property is a string of the form

*library*_name_property . *design_unit*_name_property [: body_name_property]

The string includes no leading, trailing, or embedded space characters between lexical elements. The library name property is the value of the `LibLogicalName` property of the object that represents the library containing the design unit. If the design unit is a primary unit, the design unit name property is the value of the `Name` property of the object that represents the design unit in the library information model, and the colon character and body name property are not included in the string. If the design unit is an architecture body, the design unit name property is the value of the `Name` property of the object that represents the corresponding entity declaration in the library information model, and the body name property is the simple name of the architecture body. If the design unit is a package body, the design unit name property is the value of the `Name` property of the object that represents the package declaration in the library information model, and the body name property is the letters **body** with the case of letters not specified by this standard.

19.4.6 The DefName and DefCaseName properties

Objects of class `lexicalScope` and `decl` in the library information model have both the `DefName` and `DefCaseName` properties. For a given object representing a named entity other than an anonymous named entity, the value of the `DefName` property is a string of the form

@ unit_name_property { . *lexical_scope*_name_property } [. *named_entity*_name_property]

and the value of the `DefCaseName` property is a string of the form

@ unit_name_property { . *lexical_scope*_case_name_property } [. *named_entity*_case_name_property]

The strings include no leading, trailing, or embedded space characters between lexical elements. If the named entity is a design unit, the unit name property is the value of the `UnitName` property of the given object; otherwise, the unit name property is the value of the `UnitName` property of the object that represents the design unit in which the named entity is declared. There is one lexical scope name property in the value of the `DefName` property, and one lexical scope case name property in the value of the `DefCaseName` property, for each declarative region (if any) between the design unit and the declaration of the named entity. A lexical scope name property is the value of the `Name` property of the object that represents the corresponding declarative region, and a lexical scope case name property is the value of the `CaseName` property of that object. If the named entity is a design unit, the named entity name property and the named entity case name property and the immediately preceding period characters are not included in the

strings. Otherwise, the named entity name property and the named entity case name property are the values of the `Name` and `CaseName` properties, respectively, of the object representing the named entity.

For a given object in the library information model representing an anonymous named entity, the values of the `DefName` and `DefCaseName` properties are not specified by this standard.

Objects in the design hierarchy information model that have the `Name` and `CaseName` properties also have the `DefName` and `DefCaseName` properties.

For a given object of class `decl` in the design hierarchy information model representing a named entity, the value of the `DefName` and `DefCaseName` properties are the values of the `DefName` and `DefCaseName` properties, respectively, of the object in the library information model that represents the declaration of the named entity.

For a given object of class `rootInst`, the value of the `DefName` and `DefCaseName` properties are the values of the `DefName` and `DefCaseName` properties, respectively, of the object in the library information model representing the entity declaration whose instantiation is represented by the given object.

For an object of class `packInst`, the value of the `DefName` and `DefCaseName` properties are the values of the `DefName` and `DefCaseName` properties, respectively, of the object in the library information model representing the package declaration whose instantiation is represented by the given object.

For an object of class `protectedTypeInst`, the value of the `DefName` and `DefCaseName` properties are the values of the `DefName` and `DefCaseName` properties, respectively, of the object in the library information model representing the variable declaration whose instantiation is represented by the given object.

For an object of class `concStmt` in the design hierarchy information model other than an object of class `concProcCallStmt`, or for an object of class `forLoop`, the value of the `DefName` and `DefCaseName` properties are the values of the `DefName` and `DefCaseName` properties, respectively, of the object in the library information model representing the statement whose instantiation is represented by the given object.

For an object of class `subpCall` in the design hierarchy information model representing a subprogram call, the value of the `DefName` property is a string of the form

*lexical_scope*_definition_name_property . *named_entity*_name_property

and the value of the `DefCaseName` property is a string of the form

*lexical_scope*_definition_case_name_property . *named_entity*_case_name_property

The strings include no leading, trailing, or embedded space characters between lexical elements. The lexical scope definition name property and the lexical scope definition case name property are the values of the `DefName` and `DefCaseName` properties, respectively, of the object of class `lexicalScope` in the library information model representing the declarative region immediately within which the subprogram call occurs. The named entity name property and the named entity case name property are the values of the `Name` and `CaseName` properties, respectively, of the object in the library information model representing the subprogram call.

For an object of class `indexedName` representing an element of a named entity that is a declared object of an array type, the value of the `DefName` property is a string of the form

*named_entity*_definition_name_property (literal { , literal })

and the value of the `DefCaseName` property is a string of the form

*named_entity*_definition_case_name_property (literal { , literal })

The strings include no leading, trailing, or embedded space characters between lexical elements. In the strings

- The named entity definition name property is the value of the `DefName` property of the object representing the declaration of the named entity, and the named entity definition case name property is the value of the `DefCaseName` property of that object.
- The literals are formed according to the rules for forming the literals in the `Name` and `CaseName` properties of the object (see 19.4.3).

For an object of class `selectedName` representing an element of a named entity that is a declared object of a record type, the value of the `DefName` property is a string of the form

*named_entity*_definition_name_property . *element*_simple_name

and the value of the `DefCaseName` property is a string of the form

*named_entity*_definition_case_name_property . *element*_simple_name

The strings include no leading, trailing, or embedded space characters between lexical elements. In the strings

- The named entity definition name property is the value of the `DefName` property of the object representing the declaration of the named entity, and the named entity definition case name property is the value of the `DefCaseName` property of that object.
- The element simple name is formed according to the rules for forming the element simple name in the `Name` and `CaseName` properties of the object (see 19.4.3).

For an object of class `sliceName` representing

- a slice of a named entity that is a declared object of an array type, and
- a slice in which the discrete range is in the form of a literal representing the left bound, a direction, and a literal representing the right bound,

the value of the `DefName` property is a string of the form

*named_entity*_definition_name_property (literal direction literal)

and the value of the `DefCaseName` property is a string of the form

*named_entity*_definition_case_name_property (literal direction literal)

The strings include no leading, trailing, or embedded space characters between lexical elements. In the strings

- The named entity definition name property is the value of the `DefName` property of the object representing the declaration of the object denoted by the prefix of the slice, and the named entity definition case name property is the value of the `DefCaseName` property of the former object.
- The literals and the direction are formed according to the rules for forming the literals and direction in the `Name` and `CaseName` properties of the object (see 19.4.3).

For an object of class `derefObj` in the library information model representing an element of an array variable, the values of the `DefName` and `DefCaseName` properties are strings of the same form as the

value of the DefName and DefCaseName properties, respectively, of an object of class indexedName, except that:

— The named entity definition name property is the value of the DefName property of an object representing the access value that designates the array variable, and the named entity definition case name property is the value of the DefCaseName property of that object.

— Each literal is the index value of the element for the corresponding index position of the array type of the array variable.

For an object of class derefObj in the library information model representing an element of a record variable, the values of the DefName and DefCaseName properties are strings of the same form as the value of the DefName and DefCaseName properties, respectively, of an object of class selectedName, except that the named entity definition name property is the value of the DefName property of an object representing the access value that designates the record variable, and the named entity definition case name property is the value of the DefCaseName property of that object.

For an object of class derefObj in the library information model representing a slice of an array variable, the values of the DefName and DefCaseName properties are strings of the same form as the value of the DefName and DefCaseName properties, respectively, of an object of class sliceName, except that the named entity definition name property is the value of the DefName property of an object representing the access value that designates the array variable, and the named entity definition case name property is the value of the DefCaseName property of that object.

For an object of class derefObj in the library information model representing an entire variable, denoted by a selected name with the suffix **all**, the value of the DefName property is a string of the form

*named_entity*_definition_name_property . **all**

and the value of the DefCaseName property is a string of the form

*named_entity*_case_definition_name_property . **all**

The strings include no leading, trailing, or embedded space characters between lexical elements. In the strings, the named entity definition name property is the value of the DefName property of an object representing the access value that designates the variable, and the named entity definition case name property is the value of the DefCaseName property of that object.

It is an error if a VHPI program reads the DefName or DefCaseName property of an object of class derefObj in the design hierarchy information model.

For an object of class attrName representing an attribute name, the value of the DefName property is a string of the form

*named_entity*_definition_name_property ' *attribute*_name_property

and the value of the DefCaseName property is a string of the form

*named_entity*_definition_case_name_property ' *attribute*_case_name_property

The strings include no leading, trailing, or embedded space characters between lexical elements. In the strings

— The named entity definition name property is the value of the DefName property of the object representing the prefix of the attribute name, and the named entity definition case name property is the value of the DefCaseName property of that object.

— The attribute name property and the attribute case name property are formed according to the rules for forming the attribute name property and the attribute case name property in the `Name` and `CaseName` properties of the object (see 19.4.3).

19.4.7 The FullName and FullCaseName properties

Objects of class `decl` and `name` in the library information model have the `FullName` and `FullCaseName` string properties. The value of the `FullName` property of such an object is the same as the value of the `DefName` property of the object, and the value of the `FullCaseName` property of such an object is the same as the value of the `DefCaseName` property of the object.

Objects of classes `decl`, `region`, and `name` in the design hierarchy information model have the `FullName` and `FullCaseName` string properties.

For a given object of class `decl` or `region` representing a named entity that is statically elaborated and that is either a package, declared immediately within in a package, or elaborated as a declaration in a protected type that is the type of a shared variable declared immediately within a package, the value of the `FullName` property is a string of the form

@ *library*_name_property : *package*_name_property :
[*shared_variable*_name_property :] [*named_entity*_name_property]

and the value of the `FullCaseName` property is a string of the form

@ *library*_case_name_property : *package_case*_name_property :
[*shared_variable*_case_name_property :] [*named_entity*_case_name_property]

The strings include no leading, trailing, or embedded space characters between lexical elements. The library name property and the library case name property are both the value of the `LibLogicalName` property of the object that represents the library containing the package declaration. The package name property and the package case name property are the values of the `Name` and `CaseName` properties, respectively, of the object that represents the package. The shared variable name property and the shared variable case name property are present if the given object represents a named entity elaborated as a declaration in a protected type that is the type of a shared variable declared immediately within a package. In that case, the properties are the values of the `Name` and `CaseName` properties, respectively, of the object that represents the shared variable. The named entity name property and the named entity case name property are present if the given object represents a named entity declared immediately within a package or elaborated as a declaration in a protected type that is the type of a shared variable declared immediately within a package. In that case, the properties are the values of the `Name` and `CaseName` properties, respectively, of the given object.

For a given object of class `decl` or `region` representing a named entity that is statically elaborated and that is a root design entity instance, a named entity declared in a declarative region other than immediately within a package, a named entity elaborated as a declaration in a protected type that is the type of a shared variable declared other than immediately within a package, or a concurrent statement, the value of the `FullName` property is a string of the form

: { *region*_name_property : } [*named_entity*_name_property]

and the value of the `FullCaseName` property is a string of the form

: { *region*_case_name_property : } [*named_entity*_case_name_property]

The strings include no leading, trailing, or embedded space characters between lexical elements. There is one region name property in the value of the `FullName` property, and one region case name property in the

`CaseName` property, for the root design entity and each declarative region instance in the design hierarchy between the root design entity and the named entity. Each region name property is the value of the `Name` property of the object representing the corresponding root design entity or declarative region instance, and each region case name property is the value of the `CaseName` property of the object representing the corresponding root design entity or declarative region instance. The named entity name property or named entity case name property is present if the given object represents a declared named entity. In that case, the named entity name property is the value of the `Name` property of the given object, and the named entity case name property is the `CaseName` property of the given object.

For an object of class `decl` or `region` representing a named entity that is dynamically elaborated, the value of the `FullName` property is a string of the form

*parent_process*_full_name_property : { *parent_subprogram*_full_name_property : }
[*variable*_name_property :] *named_entity*_name_property

and the value of the `FullCaseName` property is a string of the form

*parent_process*_full_case_name_property : { *parent_subprogram*_full_case_name_property : }
[*variable*_case_name_property :] *named_entity*_case_name_property

The strings include no leading, trailing, or embedded space characters between lexical elements. In the strings

— The parent process full name property is the value of the `FullName` property of the object representing the equivalent process instance from which the subprogram containing the named entity is directly or indirectly called, and the parent process full case name property is the value of the `FullCaseName` property of that object.

— There is one occurrence of the parent subprogram full name property and one occurrence of the parent subprogram full case name property for each dynamically elaborated subprogram call, if any, in the chain of subprogram calls between the equivalent process instance and the named entity. Each parent subprogram full name property is the value of the `FullName` property of the object of class `subpCall` representing the corresponding subprogram call, and each parent subprogram full case name property is the value of the `FullCaseName` property of the object of class `subpCall` representing the corresponding subprogram call.

— The variable name property and the variable case name property are present if the named entity is elaborated as a declaration in a protected type that is the type of a variable declared immediately within the subprogram, if any, that is at the end of the chain of subprogram calls leading to the named entity. In that case, the properties are the values of the `Name` and `CaseName` properties, respectively, of the object representing the elaborated variable.

— The named entity name property and the named entity case name property are the values of the `Name` and `CaseName` properties, respectively, of the object that represents the named entity.

For an object of class `indexedName` representing an element of a named entity that is a declared object of an array type, the value of the `FullName` property is a string of the form

*named_entity*_full_name_property (literal { , literal })

and the value of the `FullCaseName` property is a string of the form

*named_entity*_full_case_name_property (literal { , literal })

The strings include no leading, trailing, or embedded space characters between lexical elements. In the strings

— The named entity full name property is the value of the `FullName` property of the object representing the declaration of the named entity, and the named entity full case name property is the value of the `FullCaseName` property of that object.
— The literals are formed according to the rules for forming the literals in the `Name` and `CaseName` properties of the object (see 19.4.3).

For an object of class `selectedName` representing an element of a named entity that is a declared object of a record type, the value of the `FullName` property is a string of the form

*named_entity*_full_name_property . *element*_simple_name

and the value of the `FullCaseName` property is a string of the form

*named_entity*_full_case_name_property . *element*_simple_name

The strings include no leading, trailing, or embedded space characters between lexical elements. In the strings

— The named entity full name property is the value of the `FullName` property of the object representing the declaration of the named entity, and the named entity full case name property is the value of the `FullCaseName` property of that object.
— The element simple name is formed according to the rules for forming the element simple name in the `Name` and `CaseName` properties of the object (see 19.4.3).

For an object of class `sliceName` representing

— a slice of a named entity that is a declared object of an array type, and
— a slice in which the discrete range is in the form of a literal representing the left bound, a direction, and a literal representing the right bound,

the value of the `FullName` property is a string of the form

*named_entity*_full_name_property (literal direction literal)

and the value of the `FullCaseName` property is a string of the form

*named_entity*_full_case_name_property (literal direction literal)

The strings include no leading, trailing, or embedded space characters between lexical elements. In the strings

— The named entity full name property is the value of the `FullName` property of the object representing the declaration of the object denoted by the prefix of the slice, and the named entity full case name property is the value of the `FullCaseName` property of the former object.
— The literals and the direction are formed according to the rules for forming the literals and direction in the `Name` and `CaseName` properties of the object (see 19.4.3).

It is an error if a VHPI program reads the `FullName` or `FullCaseName` property of an object of class `derefObj` in the design hierarchy information model.

For an object of class `attrName` representing an attribute name, the value of the `FullName` property is a string of the form

*named_entity*_full_name_property ' *attribute*_name_property

and the value of the `FullCaseName` property is a string of the form

*named_entity*_full_case_name_property ' *attribute*_case_name_property

The strings include no leading, trailing, or embedded space characters between lexical elements. In the strings

— The named entity full name property is the value of the `FullName` property of the object representing the prefix of the attribute name, and the named entity full case name property is the value of the `FullCaseName` property of that object.

— The attribute name property and the attribute case name property are formed according to the rules for forming the attribute name property and the attribute case name property in the `Name` and `CaseName` properties of the object (see 19.4.3).

NOTE 1—For a named entity within a package, the value of the 'PATH_NAME attribute is a package-based path in which the library logical name may, in some designs, be the same as the root design entity name. In such designs, there may be a named entity within the design hierarchy that has the same value of the 'PATH_NAME attribute as that of a named entity within the package. The `FullName` property of the object representing the named entity in the package has the leader character replaced with "@" to avoid the ambiguity.

NOTE 2—An object of class `subpCall` in the design hierarchy information model representing a concurrent procedure call statement is treated as an object of class `region` representing a statically elaborated named entity. An object of class `subpCall` in the design hierarchy information model representing a sequential procedure call statement is treated as an object of class `region` representing a dynamically elaborated named entity.

19.4.8 The PathName and InstanceName properties

Objects of classes `decl` and `region` in the design hierarchy information model have the `PathName` and `InstanceName` string properties. The value of the `PathName` property of such an object is the same as the value of the 'PATH_NAME attribute of the named entity represented by the object, and the value of the `InstanceName` property of such an object is the same as the value of the 'INSTANCE_NAME attribute of the named entity represented by the object (see 16.2).

19.5 The stdUninstantiated package

The class diagrams in the `stdUninstantiated` package specify aspects of the VHPI information model that relate to uninstantiated design units in the VHDL model. See Figure 3, Figure 4, and Figure 5.

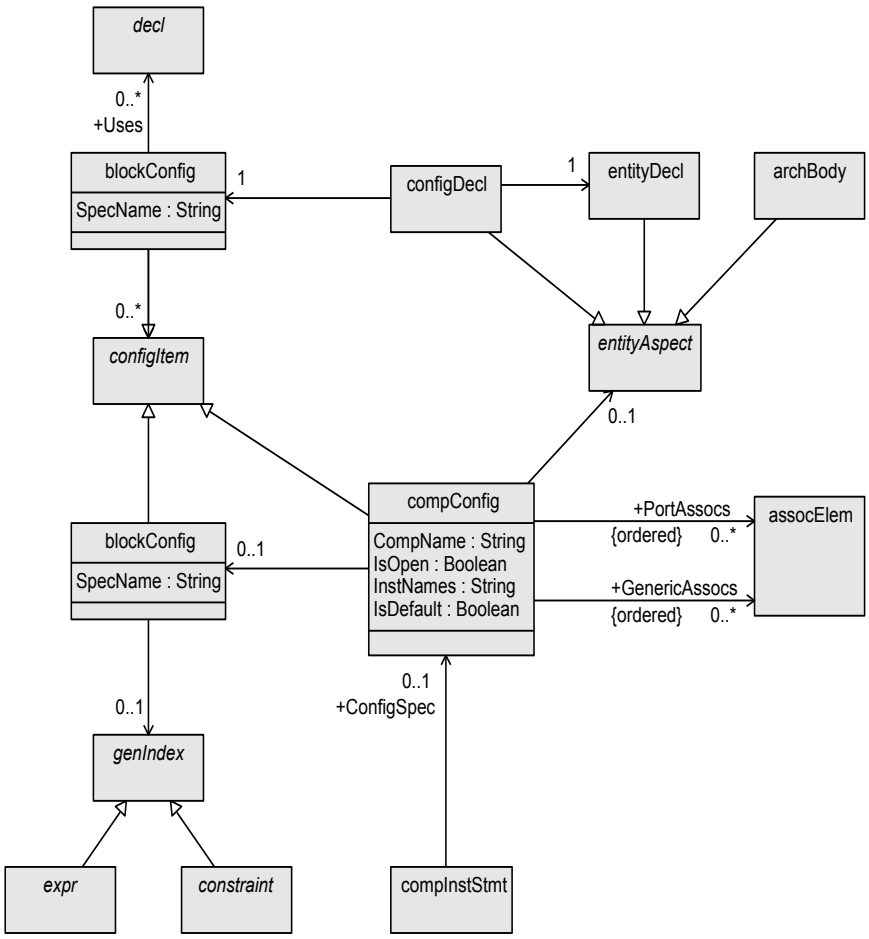

Figure 3—ConfigDecl class diagram

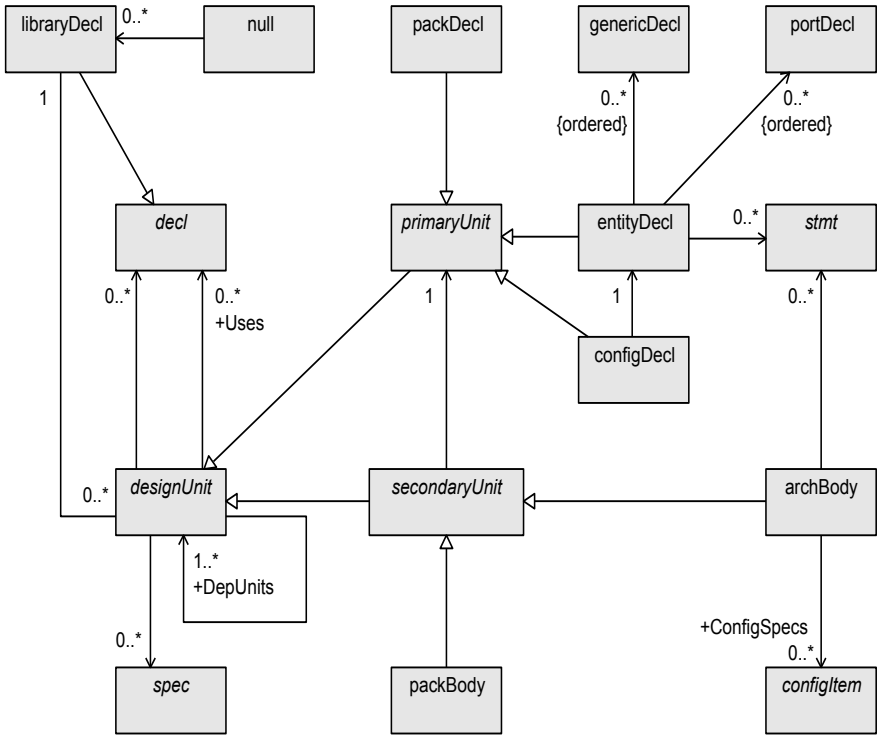

Figure 4—DesignUnit class diagram

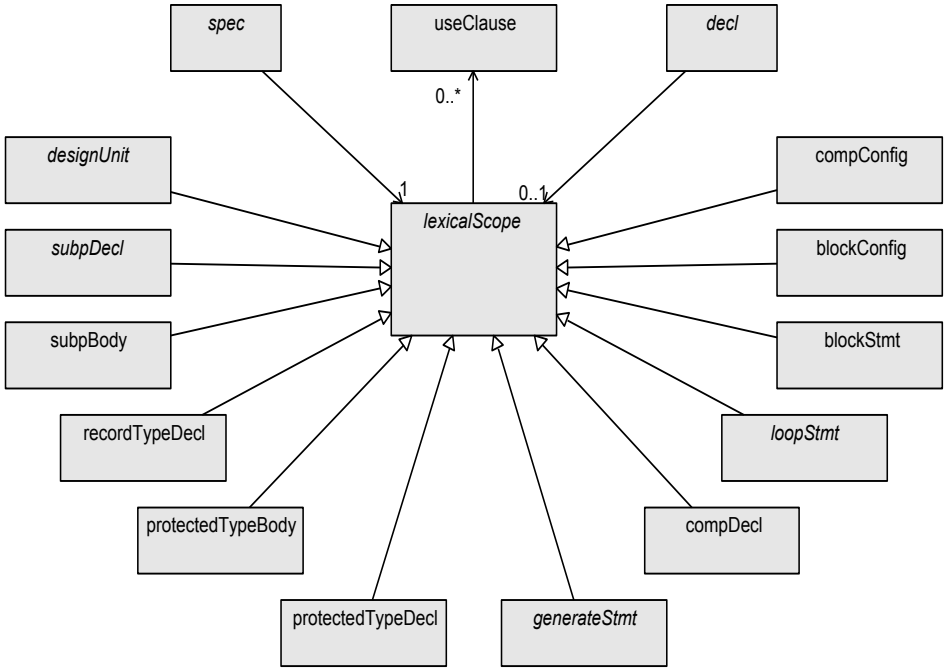

Figure 5—LexicalScope class diagram

19.6 The stdHierarchy package

The class diagrams in the `stdHierarchy` package specify aspects of the VHPI information model that relate to the VHDL design hierarchy. See Figure 6, Figure 7, Figure 8, Figure 9, Figure 10, Figure 11, Figure 12, Figure 13, Figure 14, Figure 15, Figure 16, Figure 17, and Figure 18.

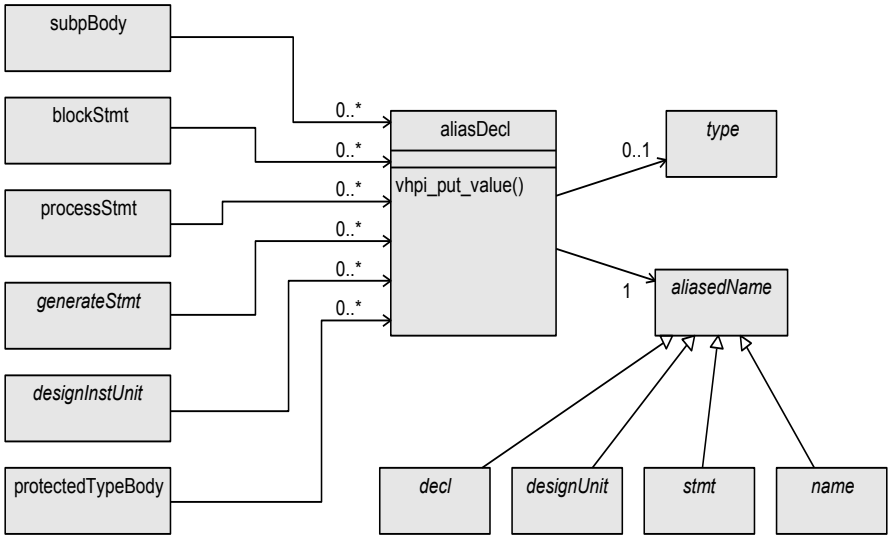

Figure 6—AliasDecl class diagram

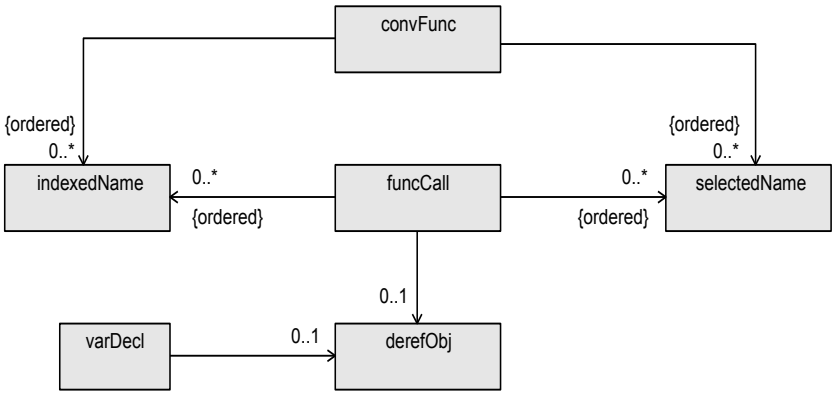

Figure 7—Composite class diagram

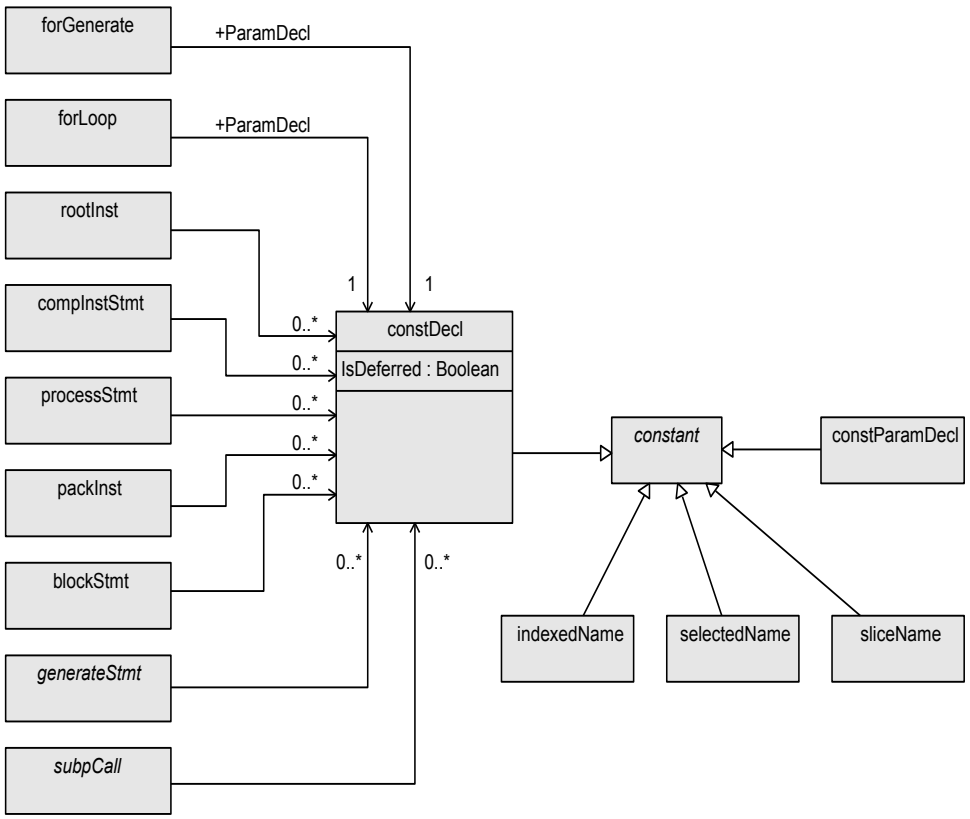

Figure 8—Constants class diagram

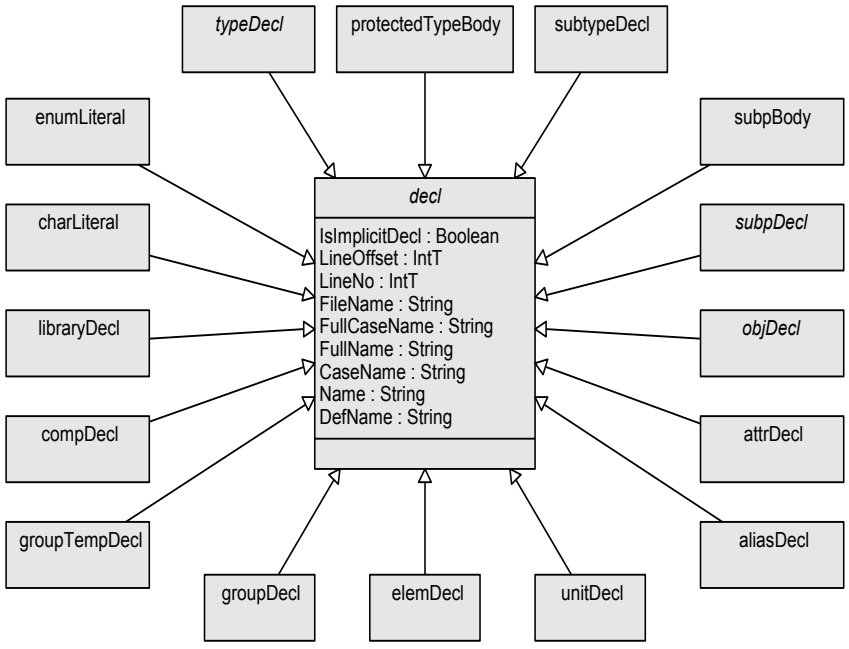

Figure 9—DeclInheritance class diagram

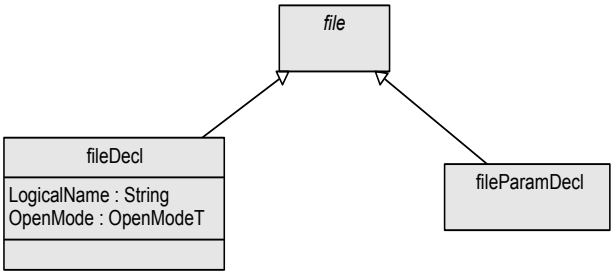

Figure 10—FileInheritance class diagram

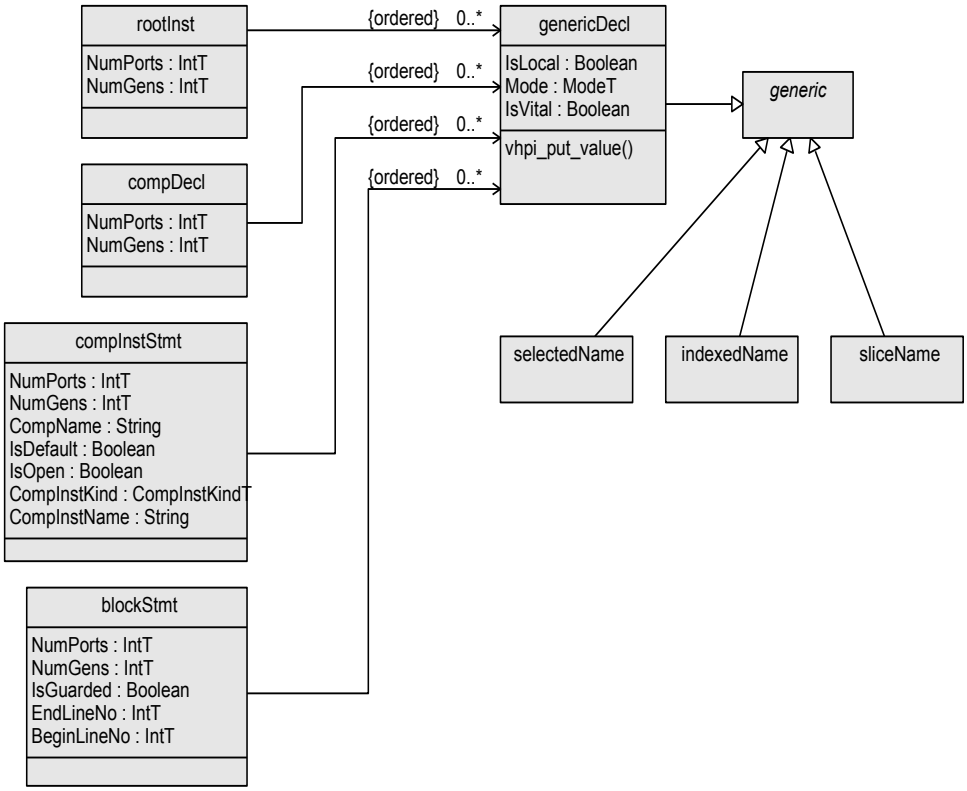

Figure 11—Generics class diagram

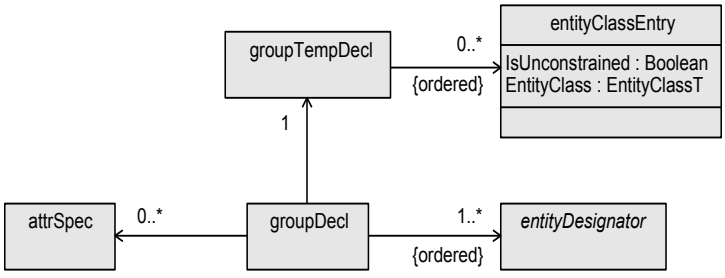

Figure 12—GroupDecl class diagram

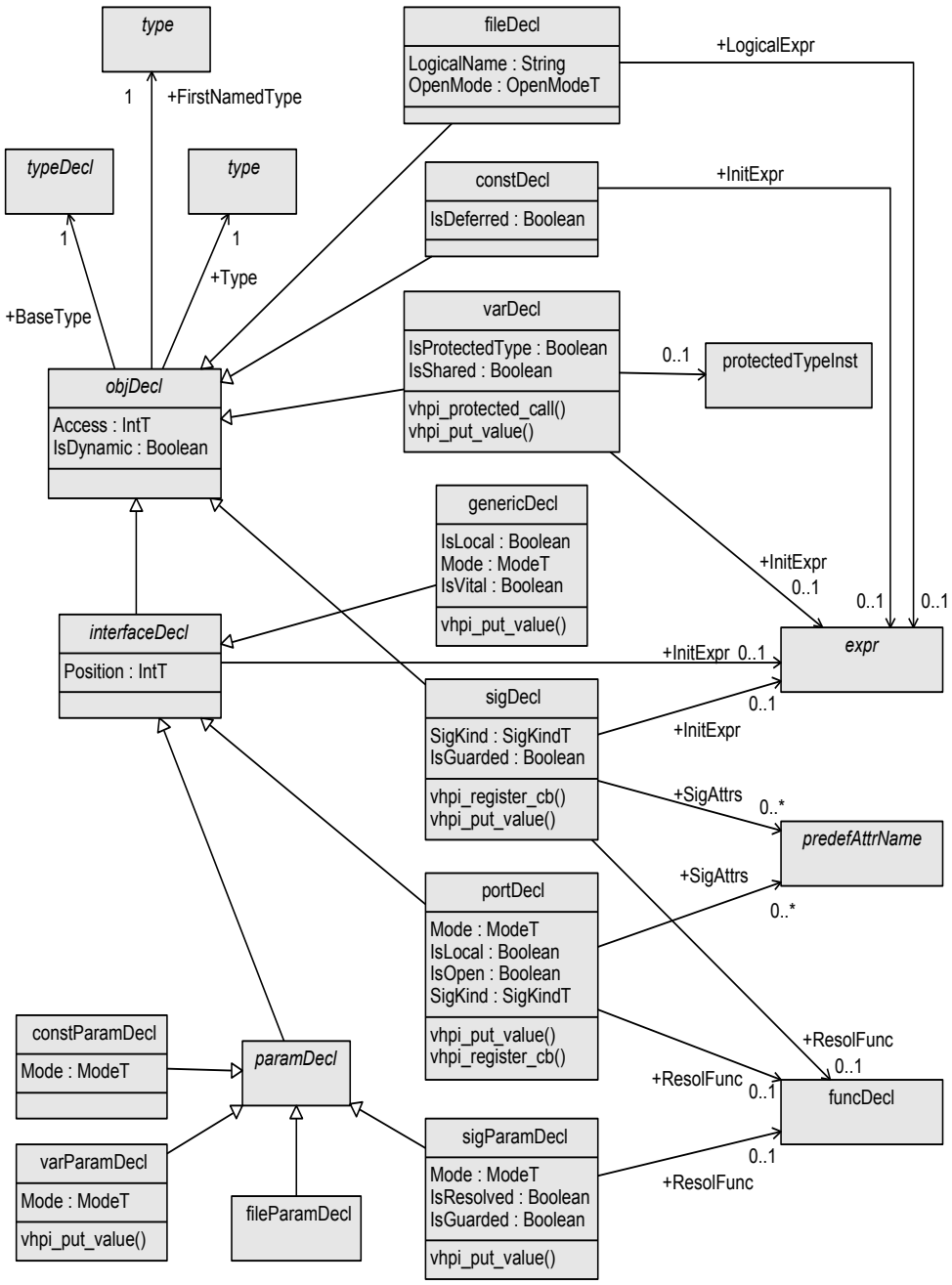

Figure 13—Object class diagram

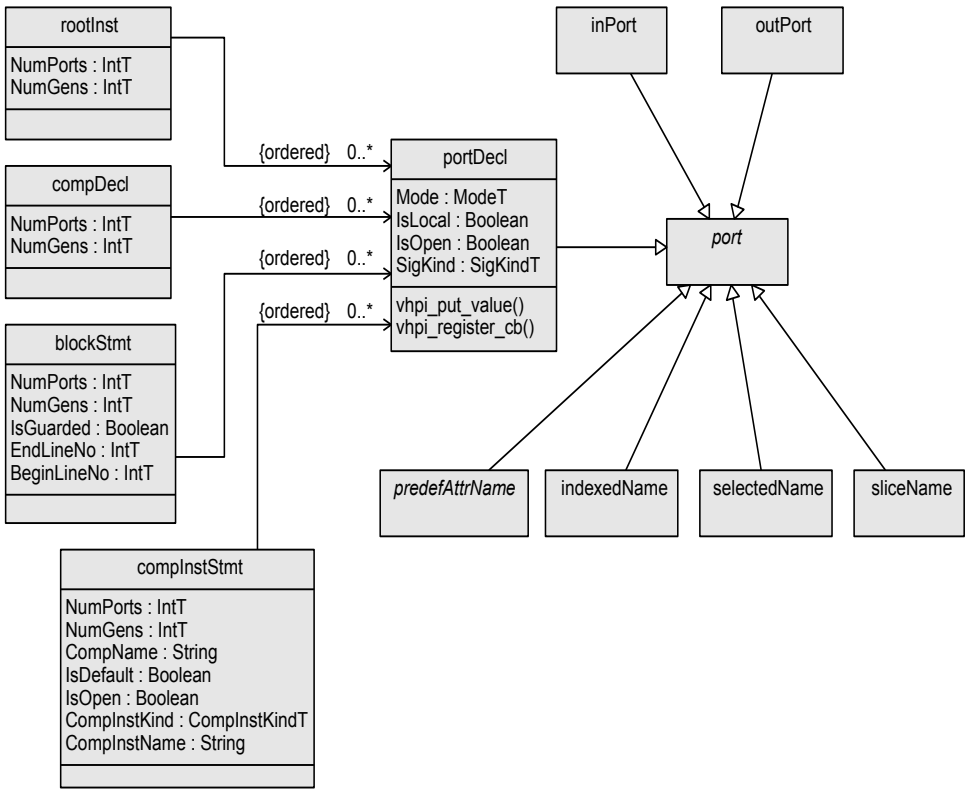

Figure 14—Ports class diagram

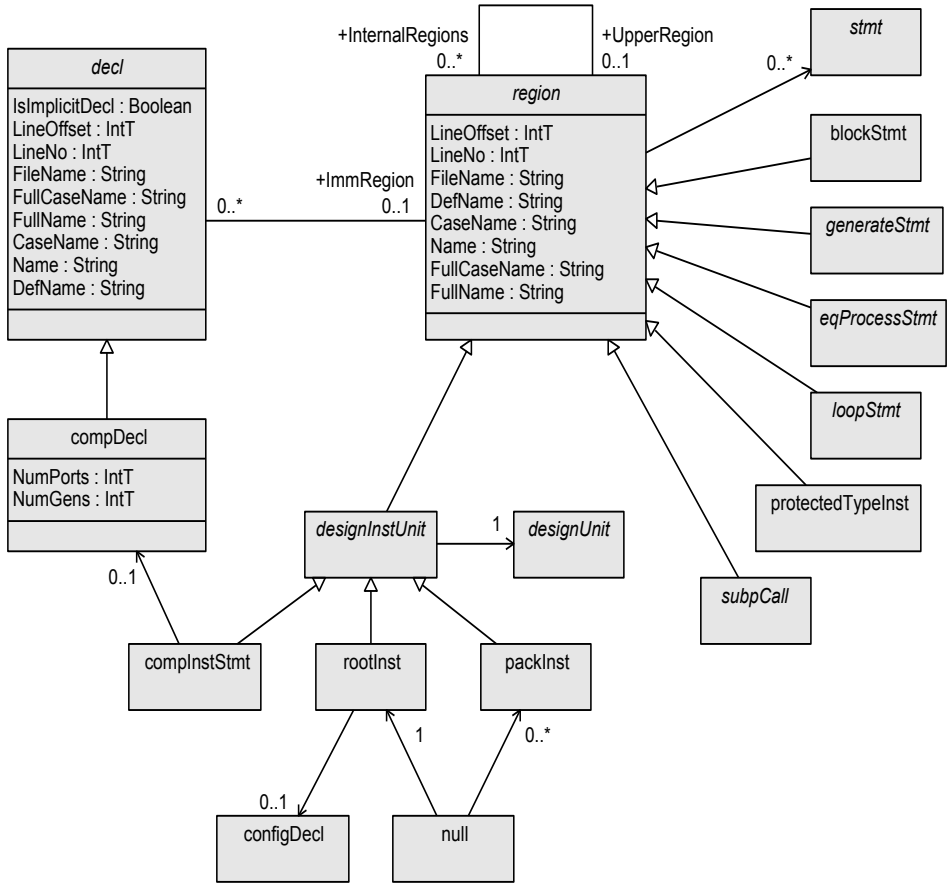

Figure 15—RegionInstance class diagram

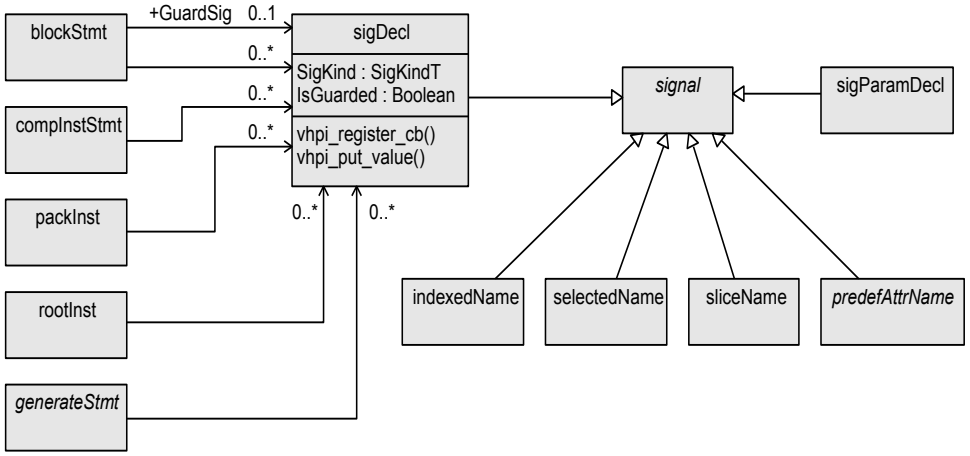

Figure 16—Signals class diagram

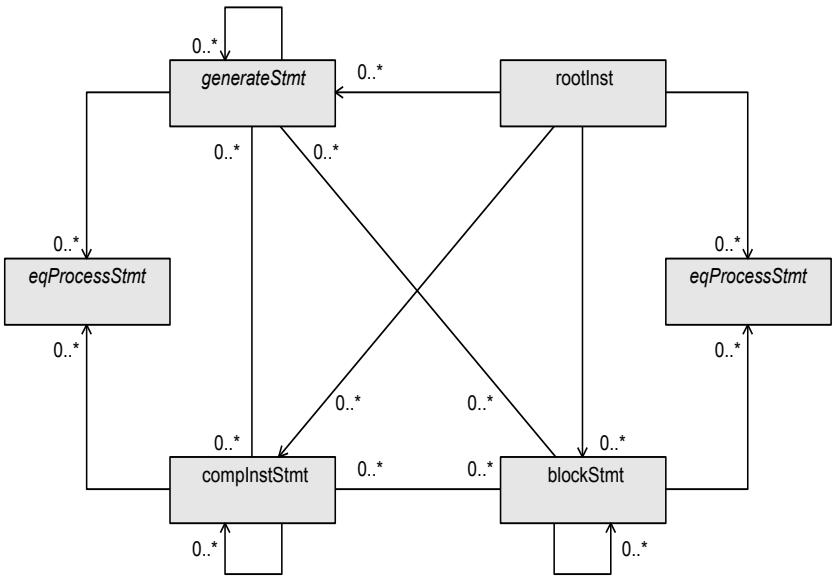

Figure 17—StructuralRegions class diagram

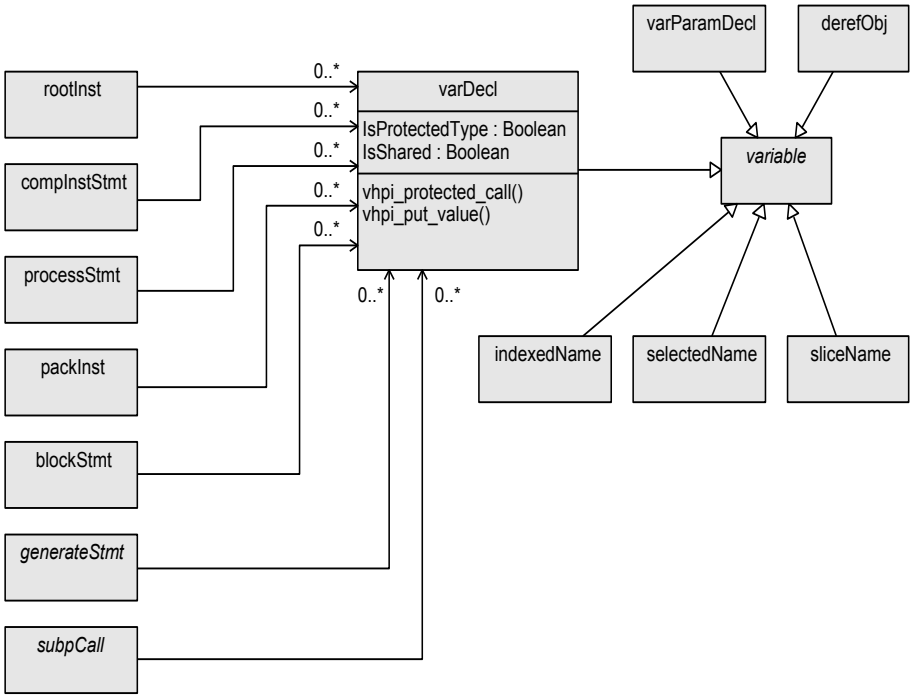

Figure 18—Variables class diagram

19.7 The stdTypes package

The class diagrams in the `stdTypes` package specify aspects of the VHPI information model that relate to types and subtypes in the VHDL model. See Figure 19, Figure 20, Figure 21, and Figure 22.

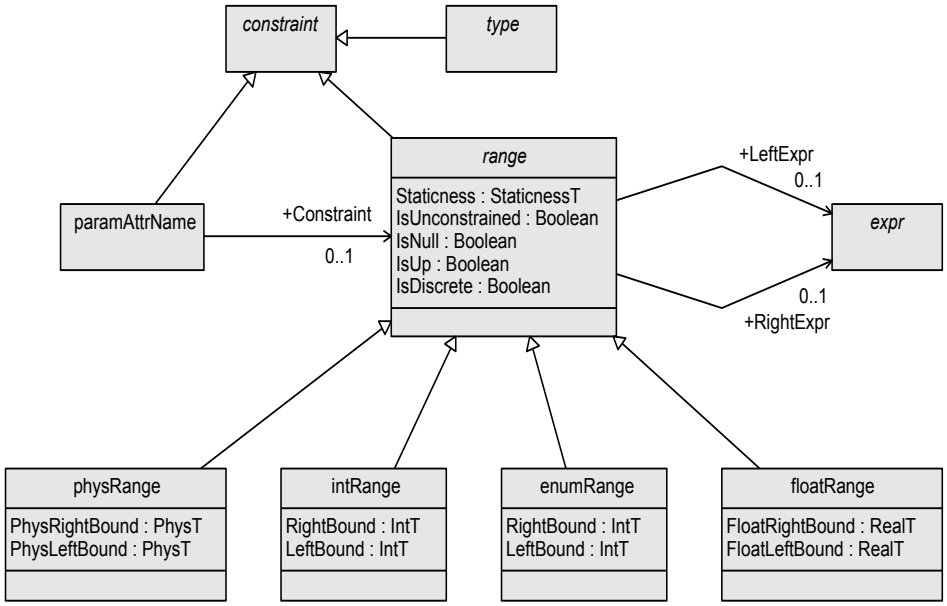

Figure 19—Constraint class diagram

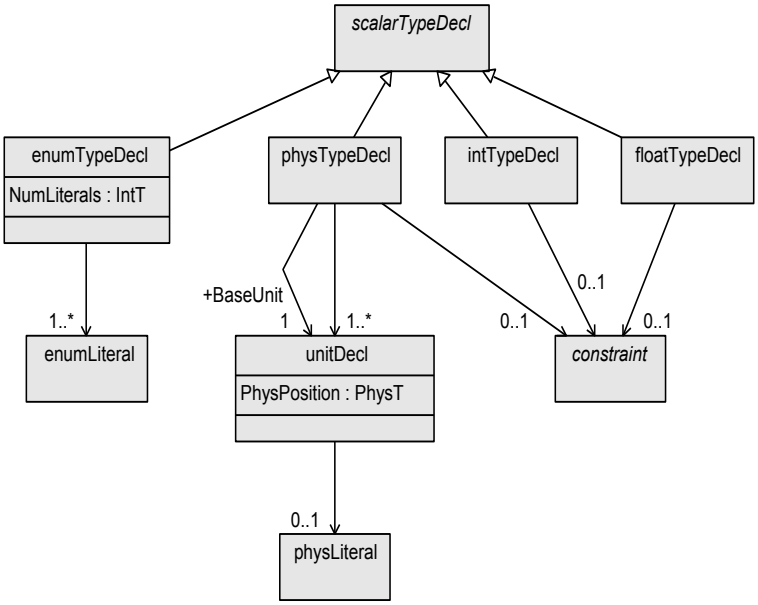

Figure 20—ScalarType class diagram

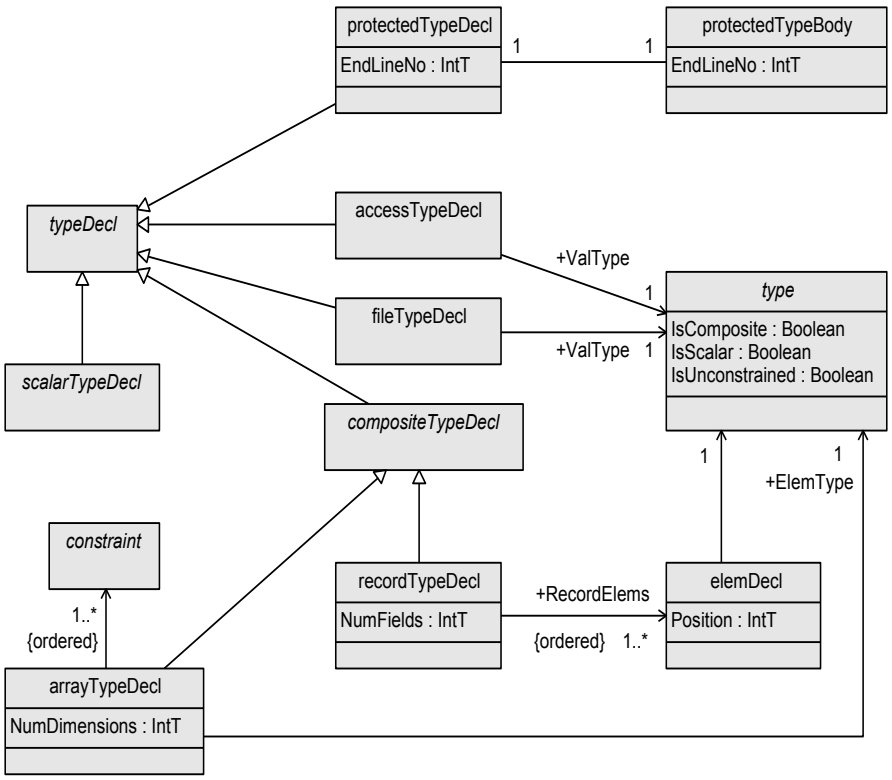

Figure 21—TypeInheritance class diagram

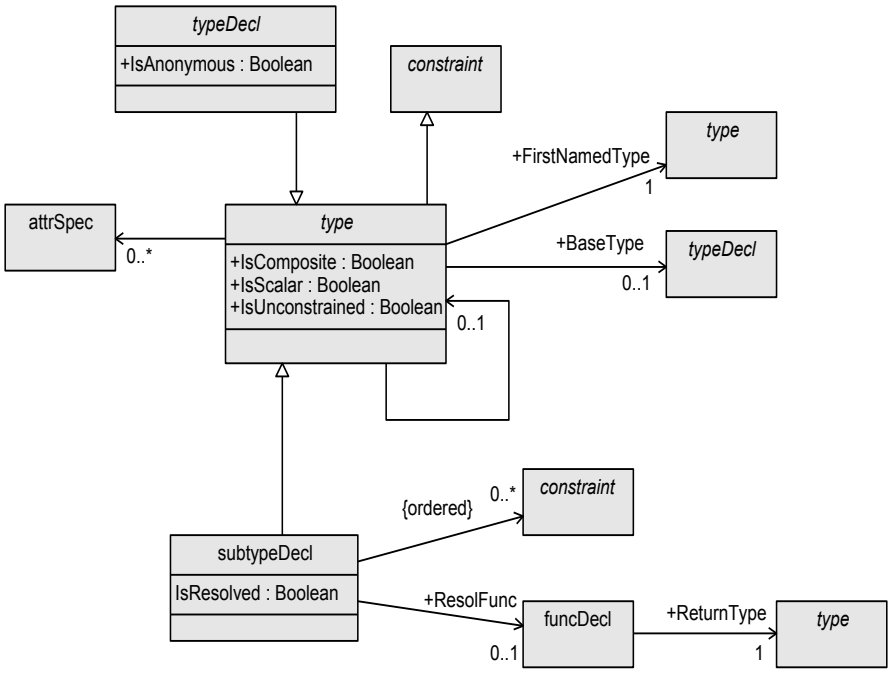

Figure 22—TypeSubtype class diagram

19.8 The stdExpr package

The class diagrams in the `stdExpr` package specify aspects of the VHPI information model that relate to expressions in the VHDL model. See Figure 23, Figure 24, Figure 25, Figure 26, Figure 27, Figure 28, and Figure 29.

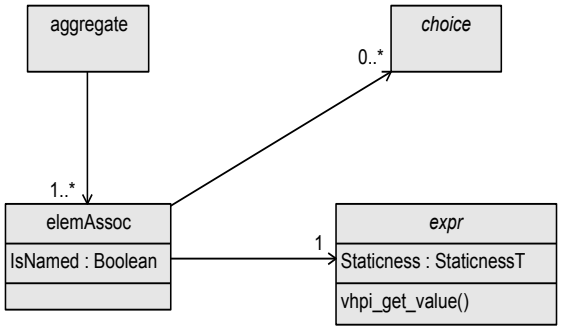

Figure 23—Aggregate class diagram

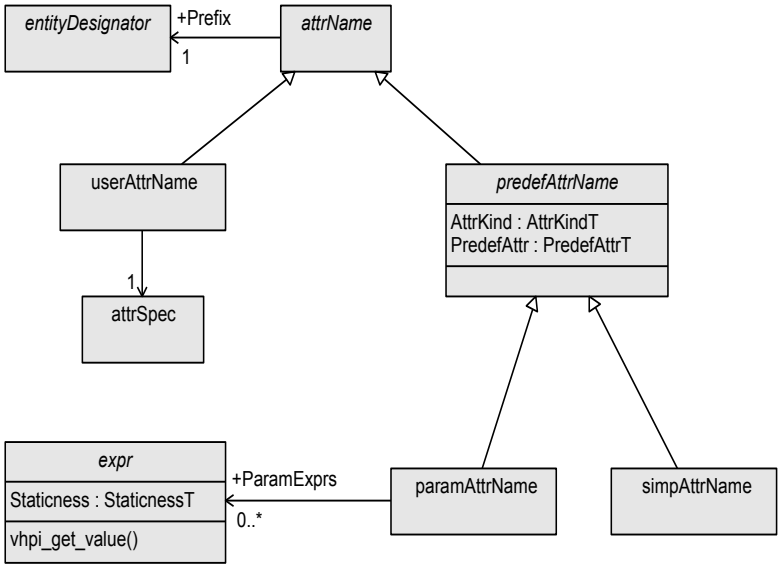

Figure 24—Attribute class diagram

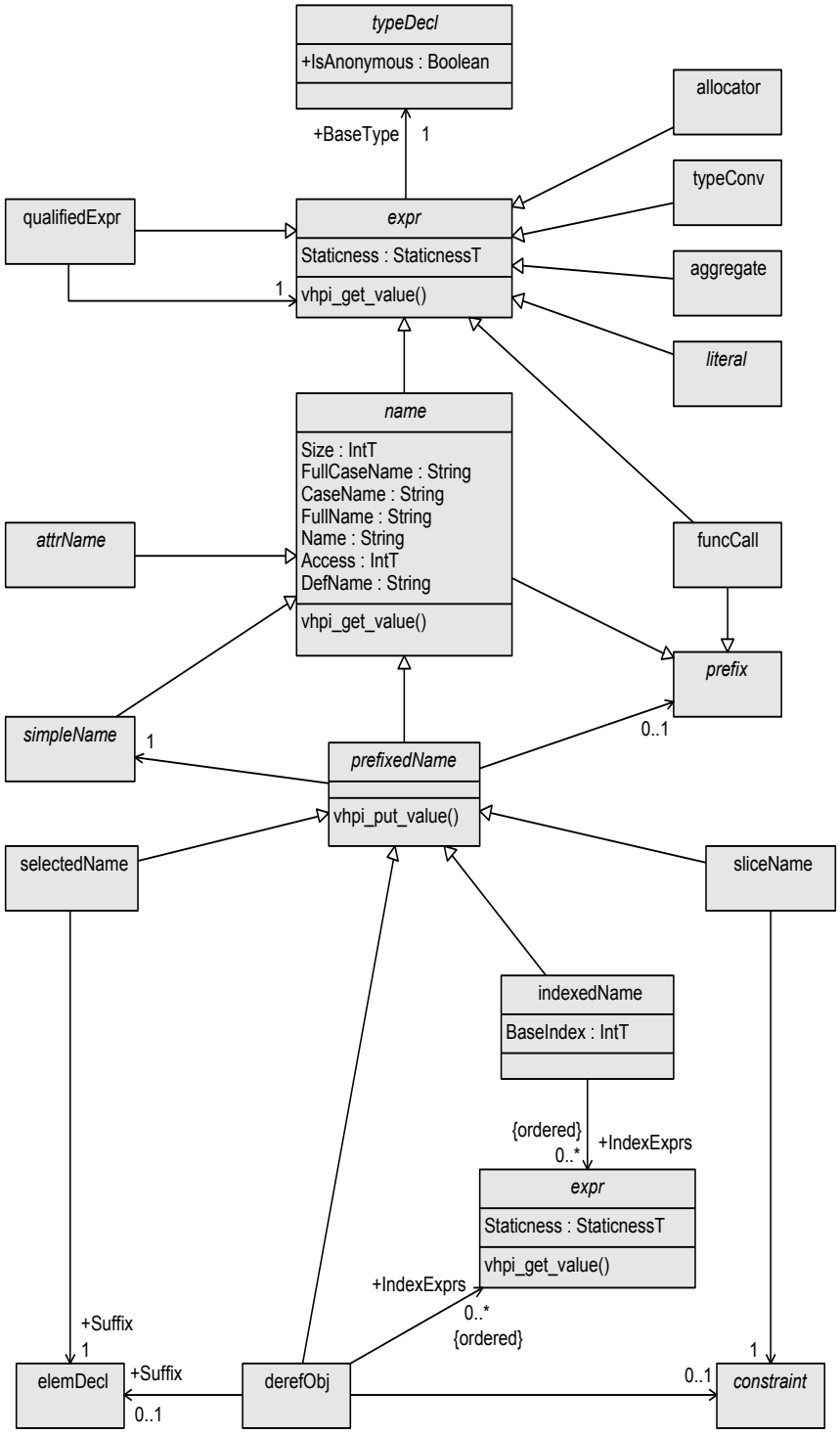

Figure 25—Expression class diagram

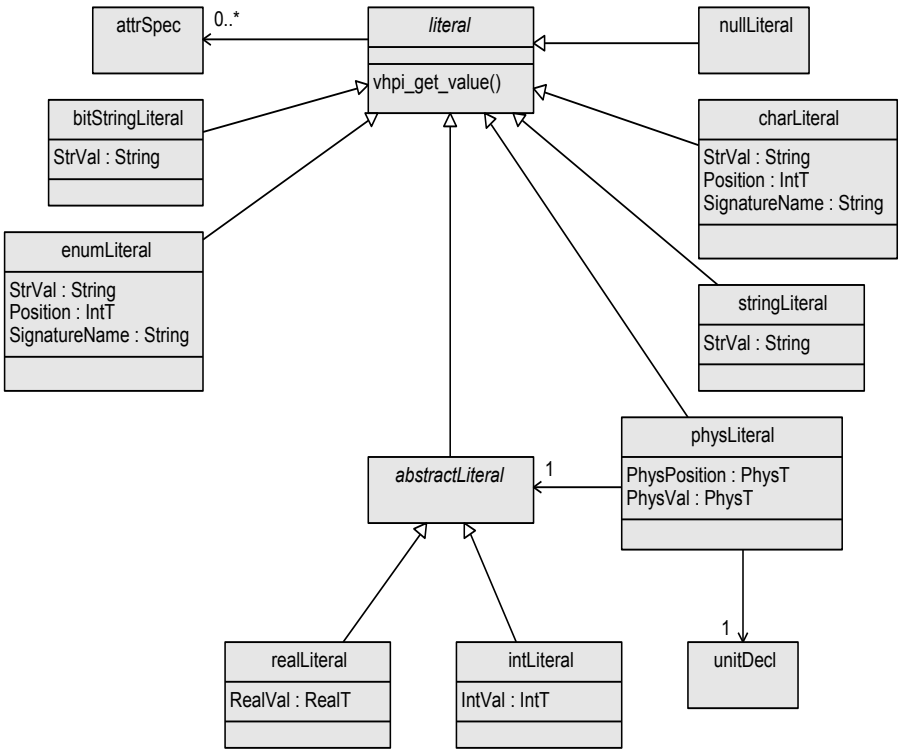

Figure 26—Literal class diagram

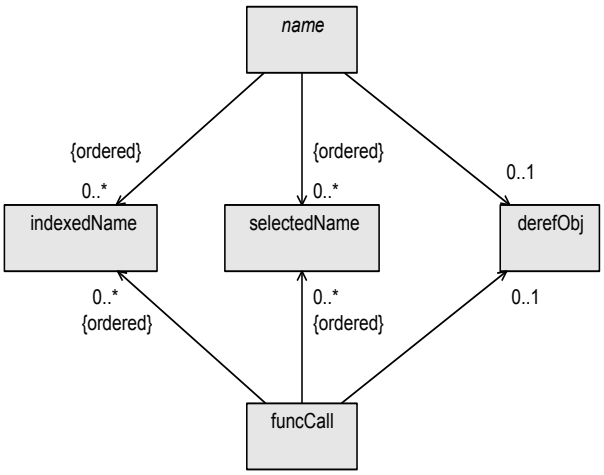

Figure 27—Name class diagram

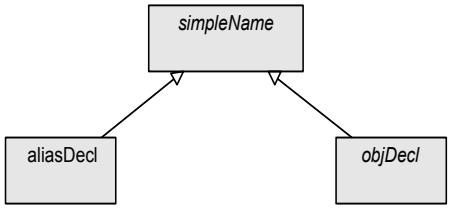

Figure 28—SimpleName class diagram

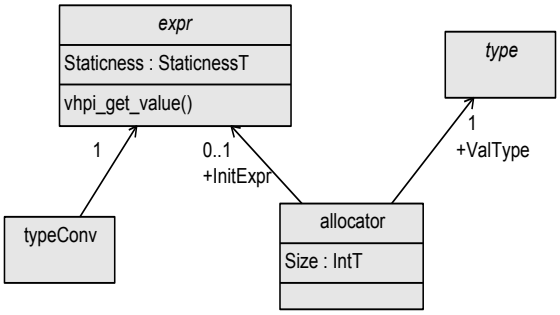

Figure 29—TypeConvAllocator class diagram

19.9 The stdSpec package

The class diagrams in the `stdSpec` package specify aspects of the VHPI information model that relate to attribute, disconnection, and configuration specifications in the VHDL model. See Figure 30, Figure 31, Figure 32, and Figure 33.

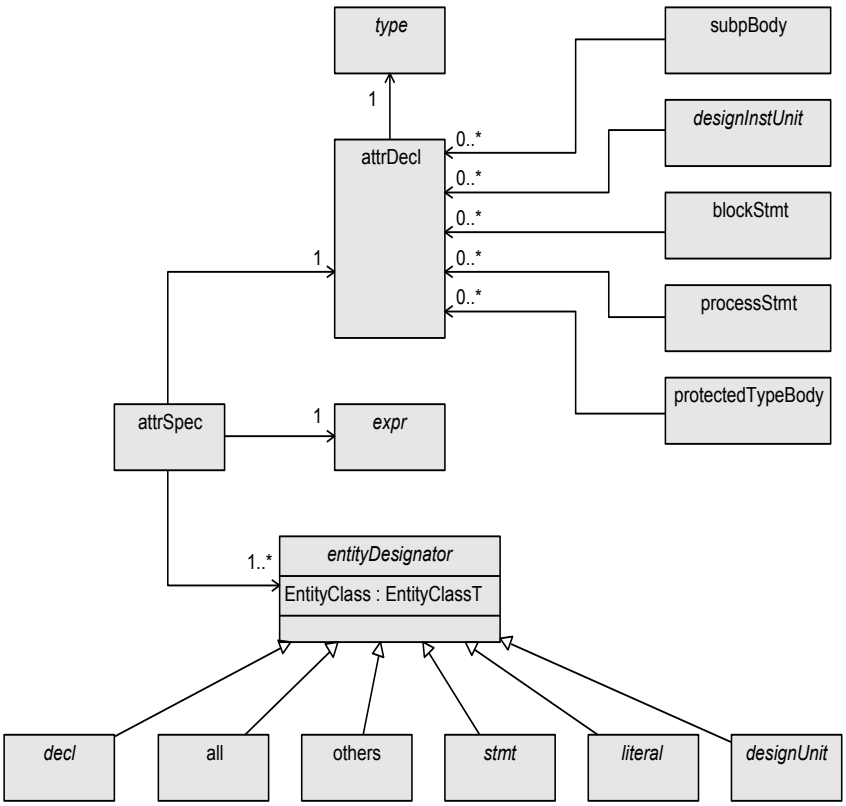

Figure 30—AttrSpec class diagram

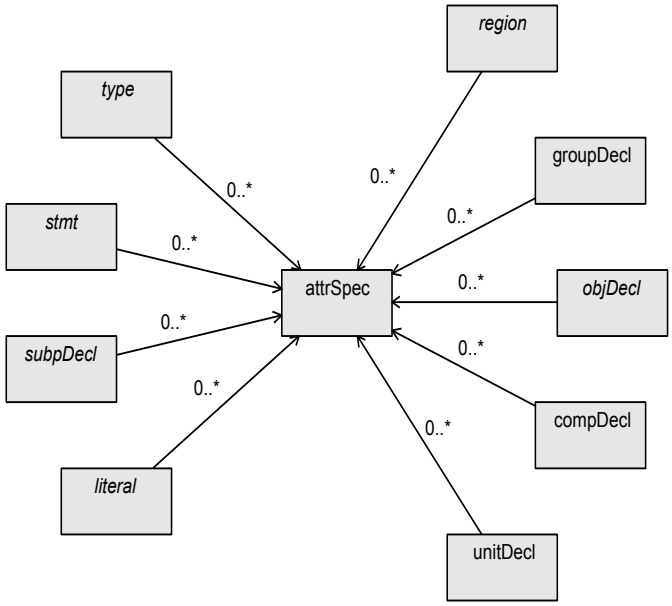

Figure 31—AttrSpecIterations class diagram

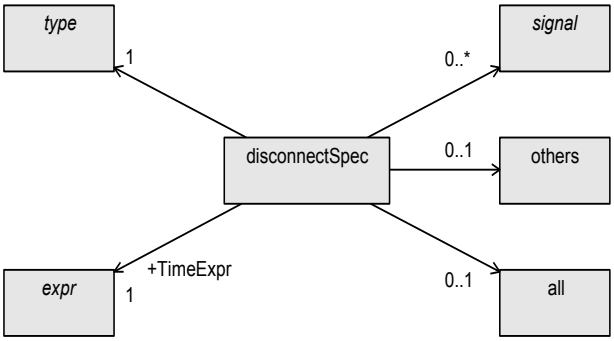

Figure 32—DisconnectionSpec class diagram

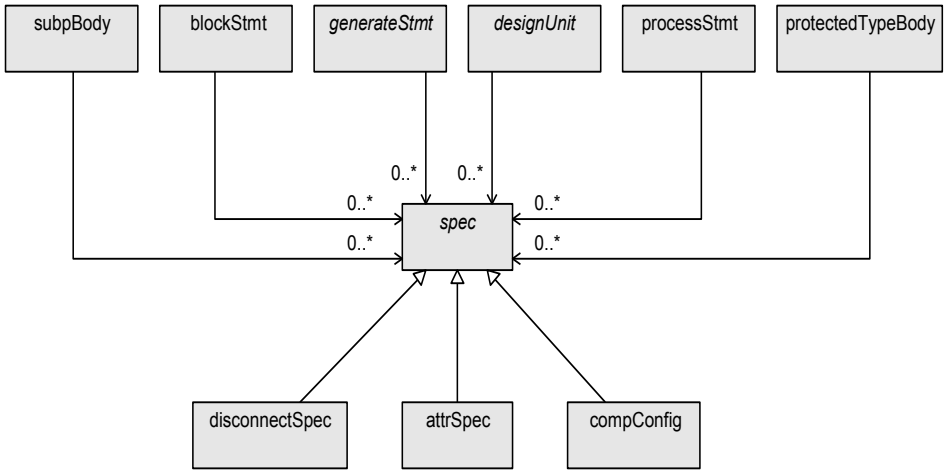

Figure 33—SpecInheritance class diagram

19.10 The stdSubprograms package

The class diagrams in the `stdSubprograms` package specify aspects of the VHPI information model that relate to subprogram declarations and subprogram calls in the VHDL model. See Figure 34 and Figure 35.

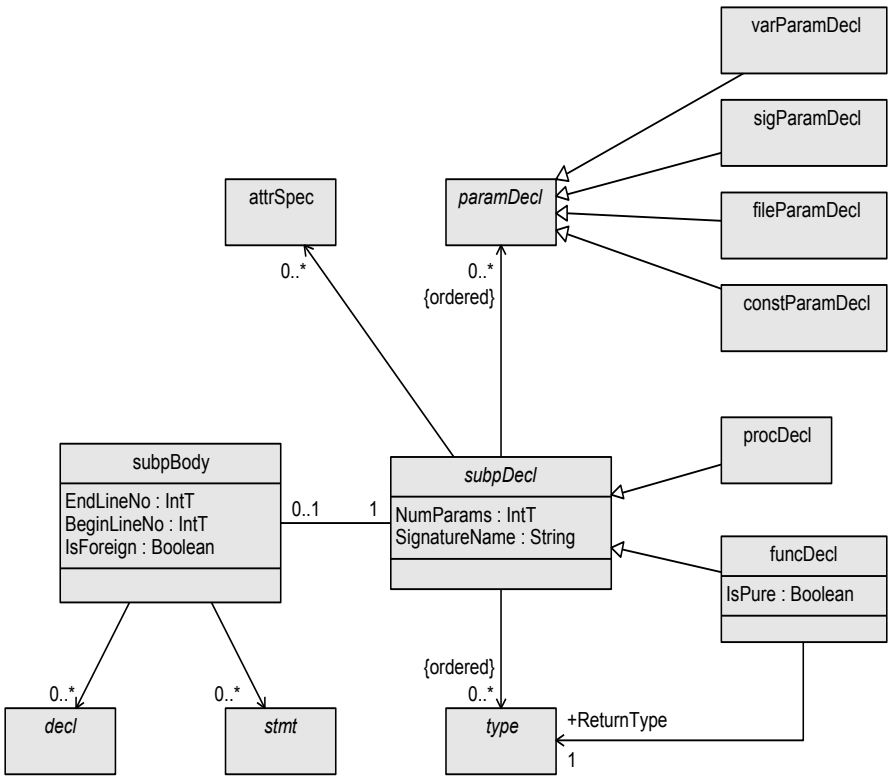

Figure 34—SubBody class diagram

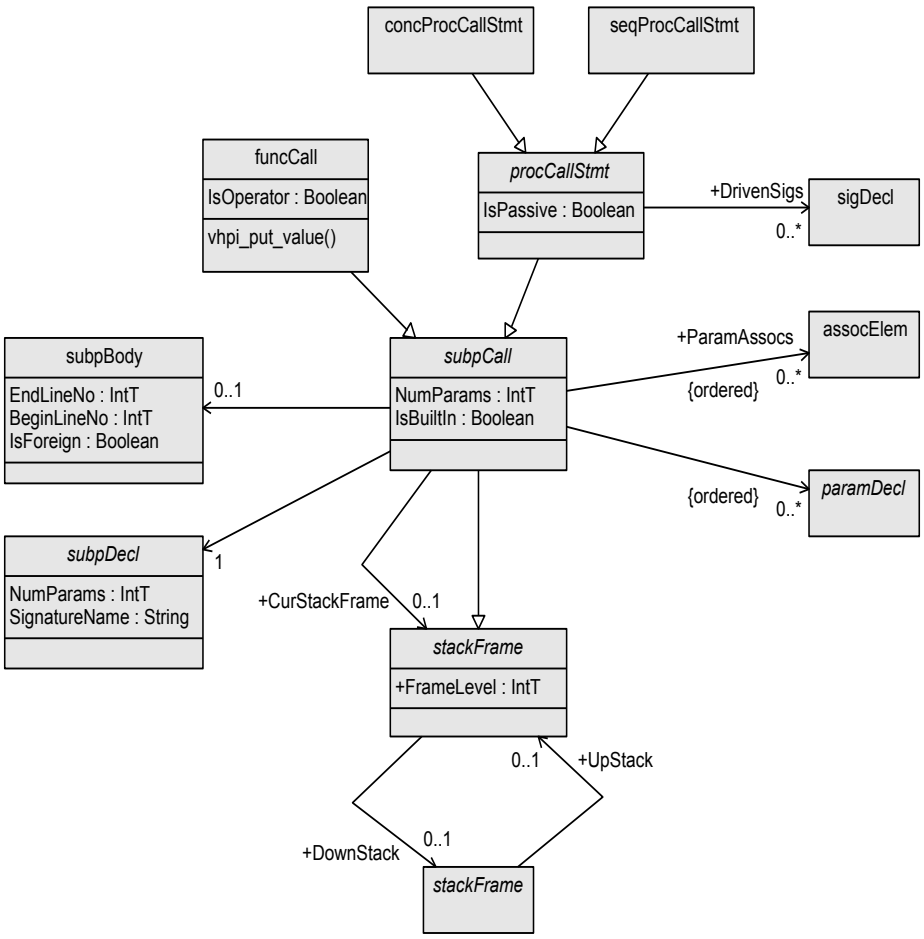

Figure 35—SubpCall class diagram

19.11 The stdStmts package

The class diagrams in the `stdStmts` package specify aspects of the VHPI information model that relate to concurrent and sequential statements in the VHDL model. See Figure 36, Figure 37, Figure 38, Figure 39, Figure 40, Figure 41, Figure 42, Figure 43, and Figure 44.

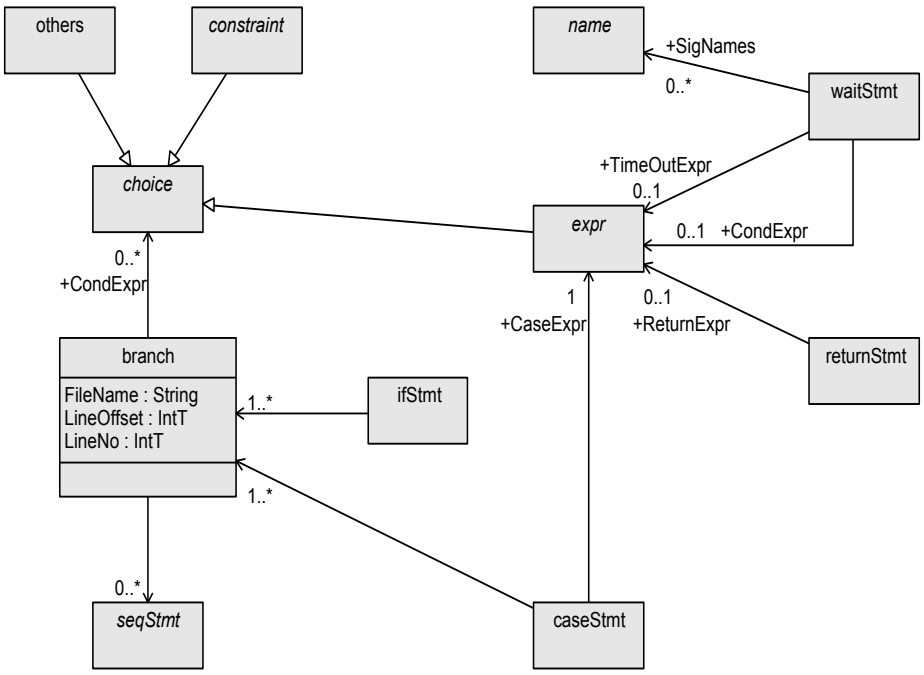

Figure 36—CaseIfWaitReturnStmt class diagram

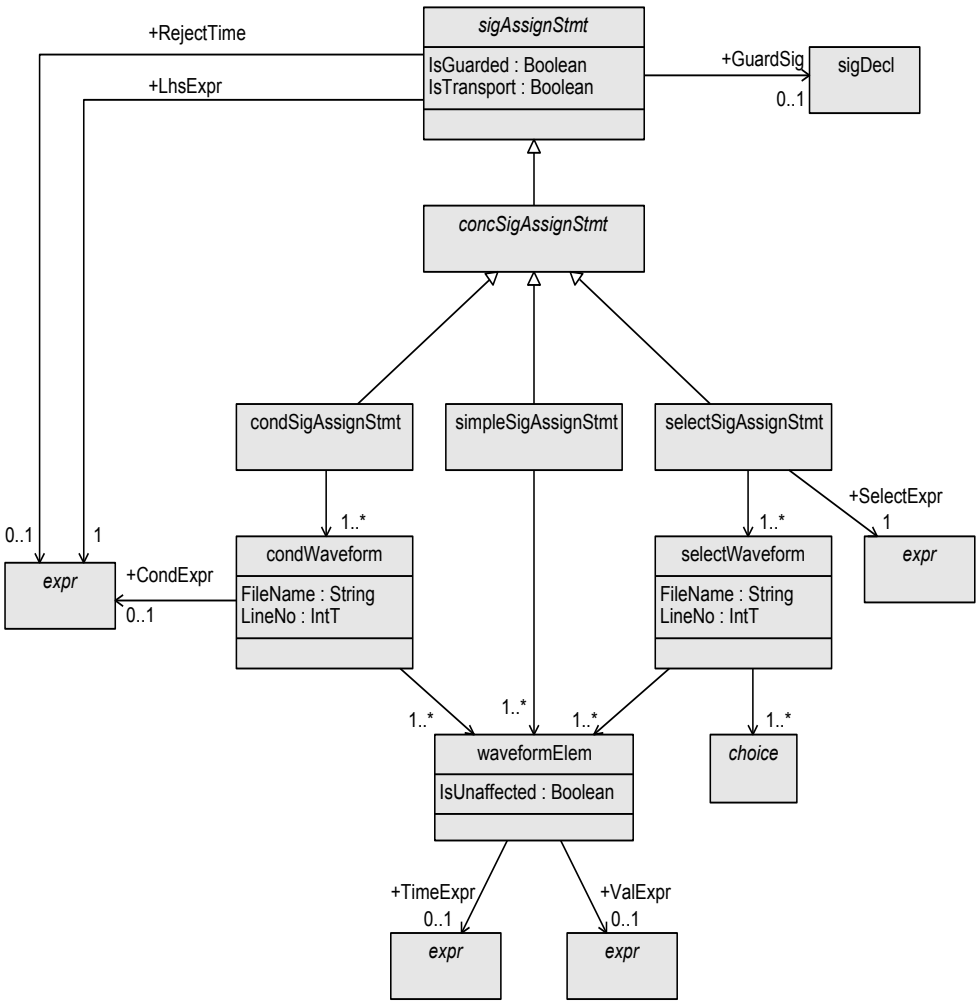

Figure 37—ConcSigAssignStmt class diagram

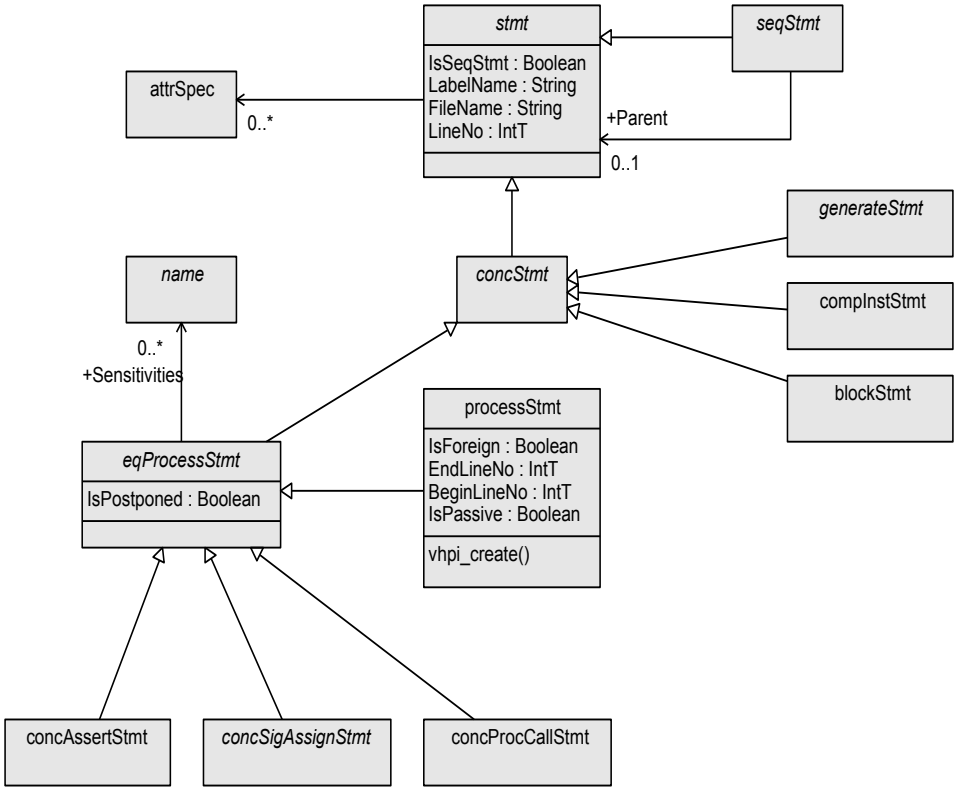

Figure 38—ConcStmt class diagram

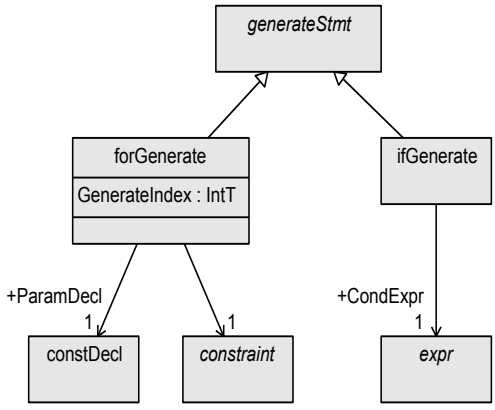

Figure 39—GenerateStmt class diagram

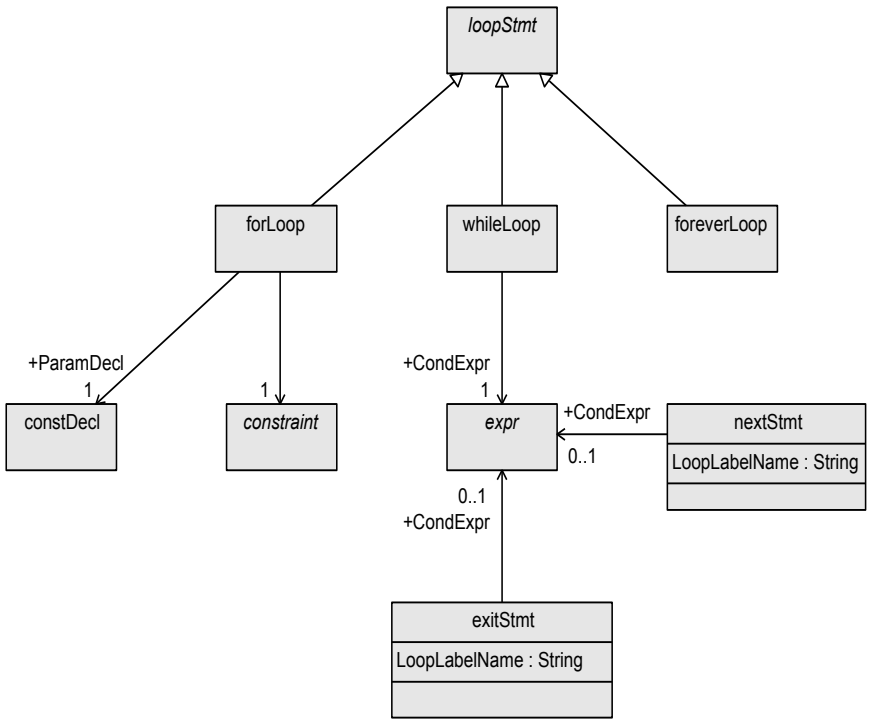

Figure 40—LoopNextStmt class diagram

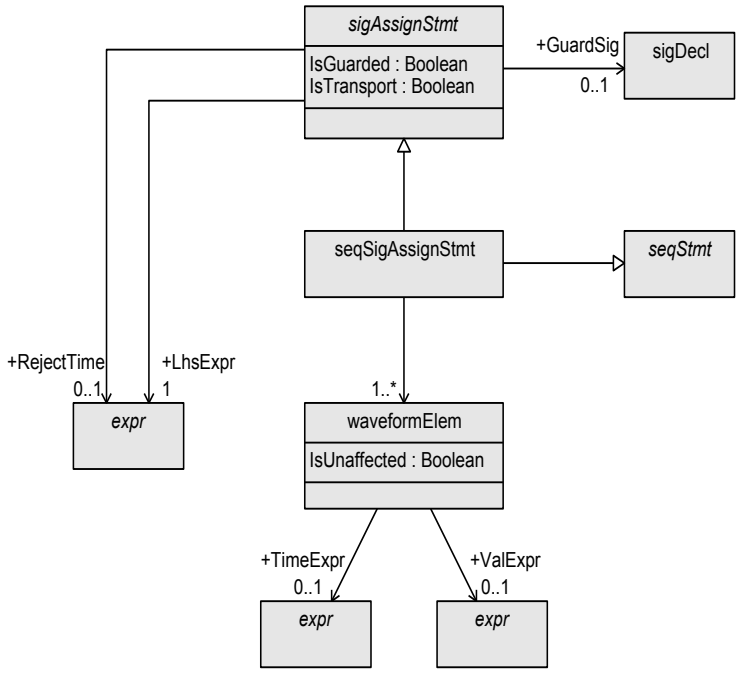

Figure 41—SeqSigAssignStmt class diagram

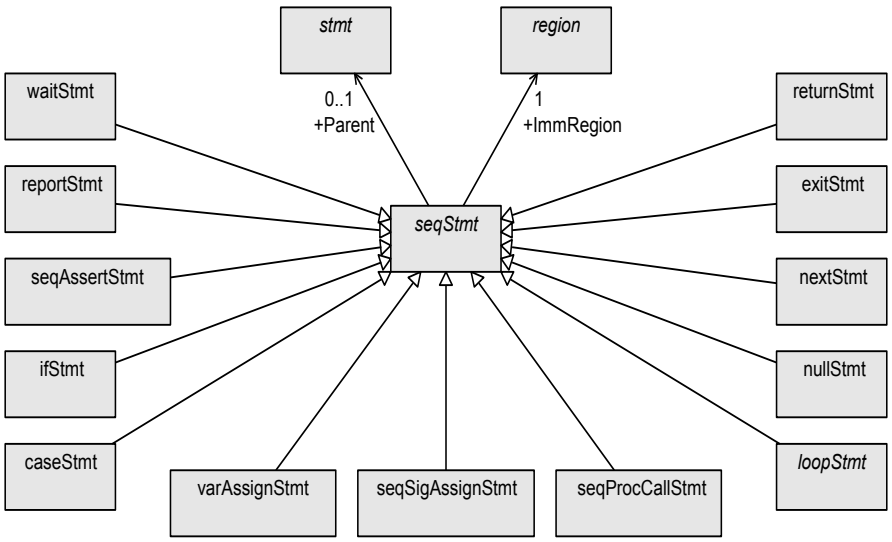

Figure 42—SeqStmtInheritance class diagram

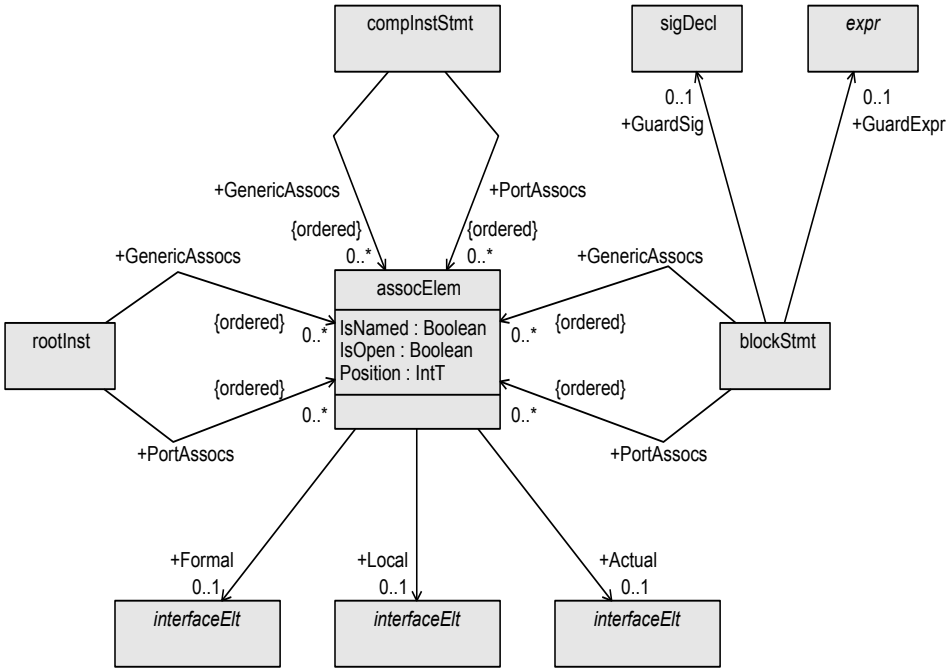

Figure 43—StructStmt class diagram

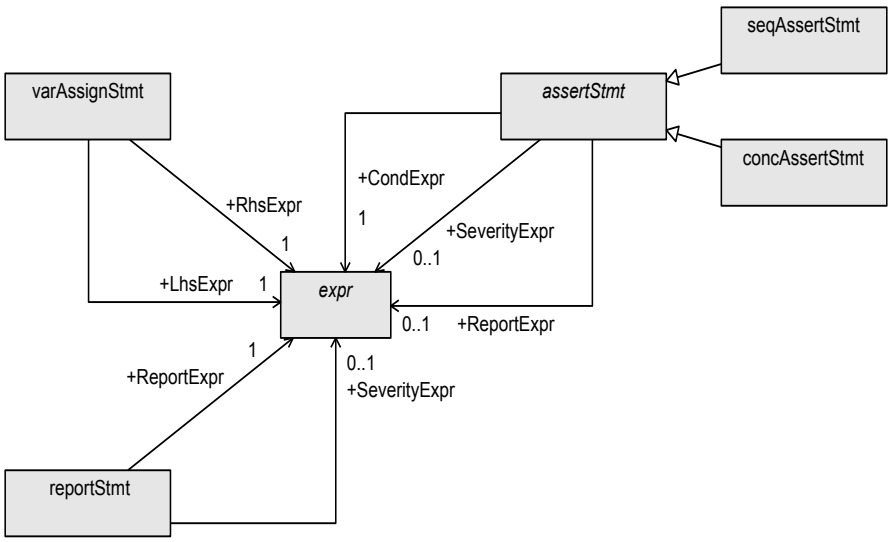

Figure 44—VarAssignAssertReportStmt class diagram

19.12 The stdConnectivity package

19.12.1 Class diagrams

The class diagrams in the `stdConnectivity` package specify aspects of the VHPI information model that relate to the interconnection of drivers, ports, and signals in the VHDL model. See Figure 45, Figure 46, Figure 47, Figure 48, and Figure 49.

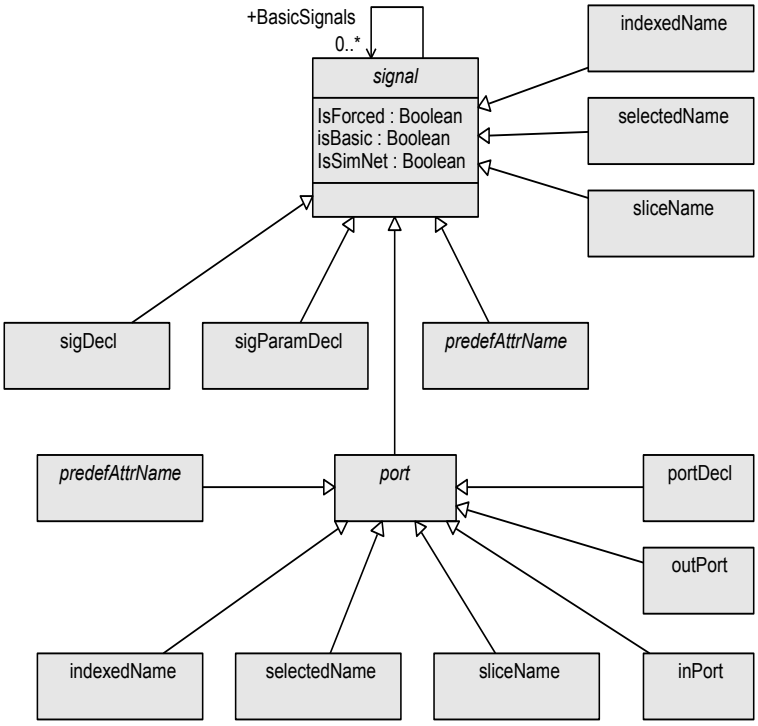

Figure 45—BasicSignal class diagram

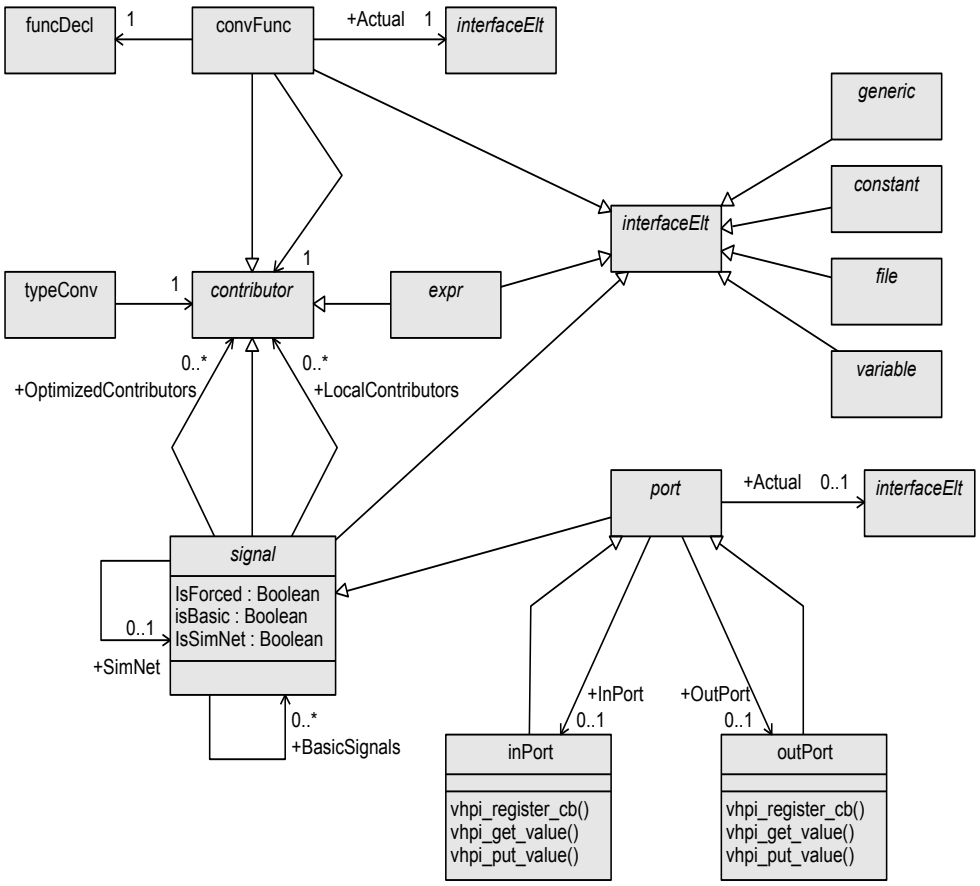

Figure 46—Connectivity class diagram

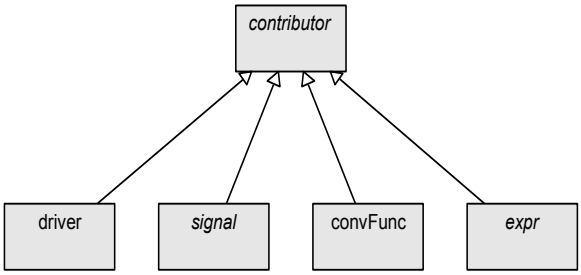

Figure 47—Contributor class diagram

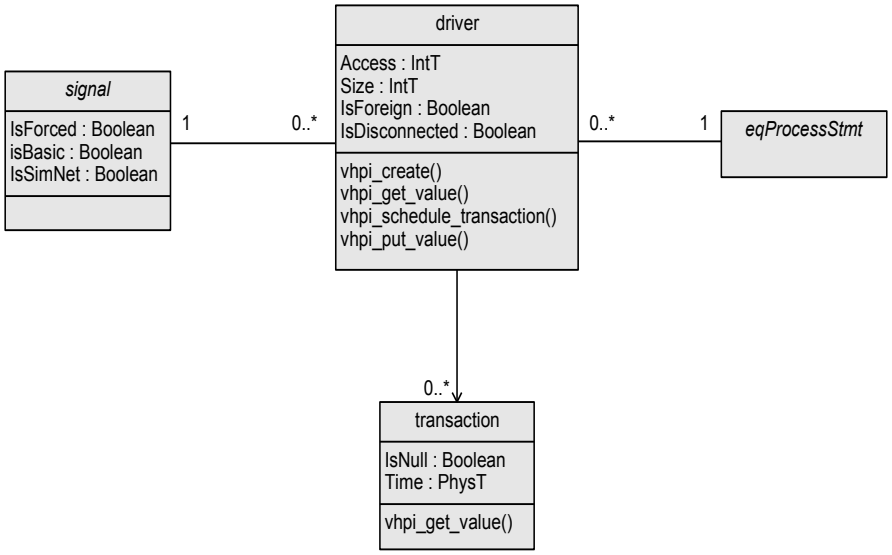

Figure 48—Driver class diagram

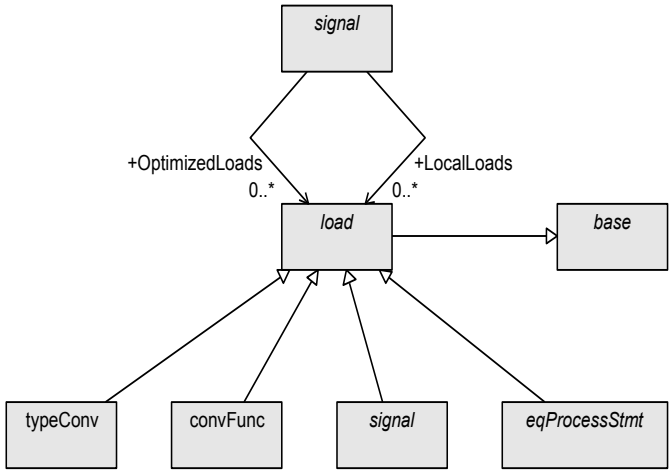

Figure 49—Loads class diagram

19.12.2 Contributors, loads, and simulated nets

19.12.2.1 General

The VHPI information model uses the class `signal` to represent parts of a net, including declared signals and ports, and subelements and slices of declared signals and ports. The information model also uses the class to represent signal parameters, implicit signals (namely, predefined attributes of signal kind and implicitly declared GUARD signals), and subelements and slices of signal parameters and implicitly declared signals.

Each basic signal, represented by an object of class `signal` for which the `IsBasic` property has the value `vhpiTrue`, has contributors and loads. A *contributor* provides a value that is used to determine the value of the object. A *load* reads the value of an object represented by an object of class signal.

The VHPI information model represents contributors and loads that are defined by the VHDL model or created by calls to the `vhpi_create` function. Such contributors and loads are called *local contributors* and *local loads*, respectively. An implementation may optimize its internal representation of contributors and loads, for example, to represent only those contributors or loads whose values are distinct. The VHPI provides associations that allow an implementation to identify such *optimized contributors* and *optimized loads*. This standard does not specify which contributors or loads, if any, are the target objects of associations that identify optimized contributors or loads.

NOTE—The VHPI information model does not represent contributors and loads for aliases of objects represented by objects of class `signal`. Those contributors and loads are represented as contributors and loads of the aliased object.

19.12.2.2 Local contributors

The local contributors for a basic signal are defined as follows:

a) For a declared signal, a port of mode **out**, the aspect of a port of mode **inout** or **buffer** that is in common with a port of mode **out**, the aspect of a port of mode **buffer** that is in common with a port of mode **in**, including a subelement or slice of any of these, each of the following is a local contributor:
   - A driver of the signal, represented by an object of class `driver`
   - A port of mode **out**, **inout**, or **buffer**, represented by an object of class `interfaceElt`, with which the signal is associated as an actual in an association element in which the formal part is in the form of the port name
   - A type conversion, represented by an object of class `typeConv`, or a conversion function call, represented by an object of class `convFunc`, occurring as the formal part of an association element in which the signal name is the actual designator
   - If the signal has no sources, the default expression, represented by an object of class `expr`, in the declaration of the signal

b) For a port of mode **in**, or the aspect of a port of mode **inout** that is in common with a port of mode **in**, including a subelement or slice of any of these, each of the following is a local contributor:
   - If the port is associated with an actual object or expression in an association element in which the actual part is the name of the actual object or is an expression, the actual object, or expression, represented by an object of class `expr`
   - If the port is associated with an actual object in an association element in which the actual part is in the form of a type conversion or a conversion function call, the type conversion, represented by an object of class `typeConv`, or the conversion function call, represented by an object of class `convFunc`, respectively
   - If the port is unassociated or unconnected and the declaration of the port includes a default expression, the default expression, represented by an object of class `expr`

c) For a formal signal parameter that is associated with an actual signal that is a basic signal, each of the following is a local contributor:
   - If the formal signal parameter is of mode **in** or **inout**, the local contributors of the actual signal
   - If the formal signal parameter is of mode **out**, the driver for the formal signal parameter

NOTE—A signal that is one of the predefined attributes 'DELAYED, 'STABLE, 'QUIET, or 'TRANSACTION may be a contributor.

19.12.2.3 Local loads

The local loads for a basic signal are defined as follows:

a) A process, or a concurrent statement that is equivalent to a process, represented by an object of class `eqProcessStmt`, that reads the basic signal or an alias of the basic signal

b) A port of mode **in** or **inout**, represented by an object of class `interfaceElt`, with which the basic signal is associated as an actual in an association element in which the actual part is in the form of the name of the basic signal

c) A type conversion, represented by an object of class `typeConv`, or a conversion function call, represented by an object of class `convFunc`, occurring as the actual part of an association element in which the name of the basic signal is the actual designator

d) For a basic signal that is a port of mode **out** or for the aspect of a basic signal that is a port of mode **inout** or **buffer** that is in common with a port of mode **out**, where the port is associated in an association element with an actual object

   — If the formal part of the association element is the name of the port, the actual object

   — If the formal part is in the form of a type conversion or a conversion function call, the type conversion, represented by an object of class `typeConv`, or the conversion function call, represented by an object of class `convFunc`, respectively

19.12.2.4 Simulated nets

Where a number of objects represented by objects of class `signal` have the same effective and driving values, as appropriate, at all simulations times, those objects jointly form a *simulated net*. An implementation may represent a simulated net by selecting one of the constituent objects as a representative of the simulated net, setting the value of its `IsSimNet` property to the value `vhpiTrue` and making it the target object of the `SimNet` association for each object in the simulated net; for the remaining objects, the implementation sets the value of the `IsSimNet` property to the value `vhpiFalse`.

19.13 The stdCallbacks package

The `stdCallbacks` package contains the `Callbacks` class diagram that specifies aspects of the VHPI information model that relate to callbacks in VHPI programs. See Figure 50.

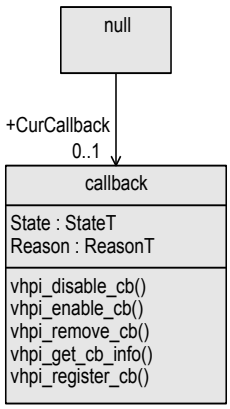

Figure 50—Callbacks class diagram

19.14 The stdEngine package

The `stdEngine` package contains the `SimulatorKernel` class diagram that specifies aspects of the VHPI information model that relate to the simulation kernel. See Figure 51.

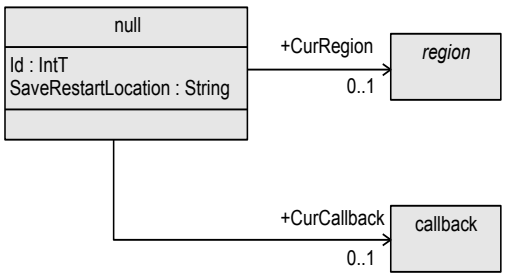

Figure 51—SimulatorKernel class diagram

19.15 The stdForeign package

The `stdForeign` package contains the `ForeignModel` class diagram that specifies aspects of the VHPI information model that relate to foreign models and applications implemented by VHPI programs. See Figure 52.

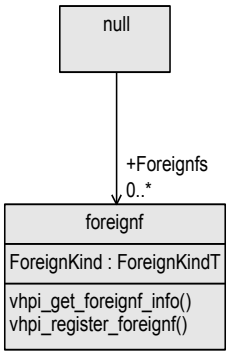

Figure 52—ForeignModel class diagram

19.16 The stdMeta package

The class diagrams in the `stdMeta` package specify aspects of the VHPI information model that relate to the VHPI tool, collections, and iterators. The package also contains a class diagram that relates classes to the `base` class. See Figure 53, Figure 54, and Figure 55.

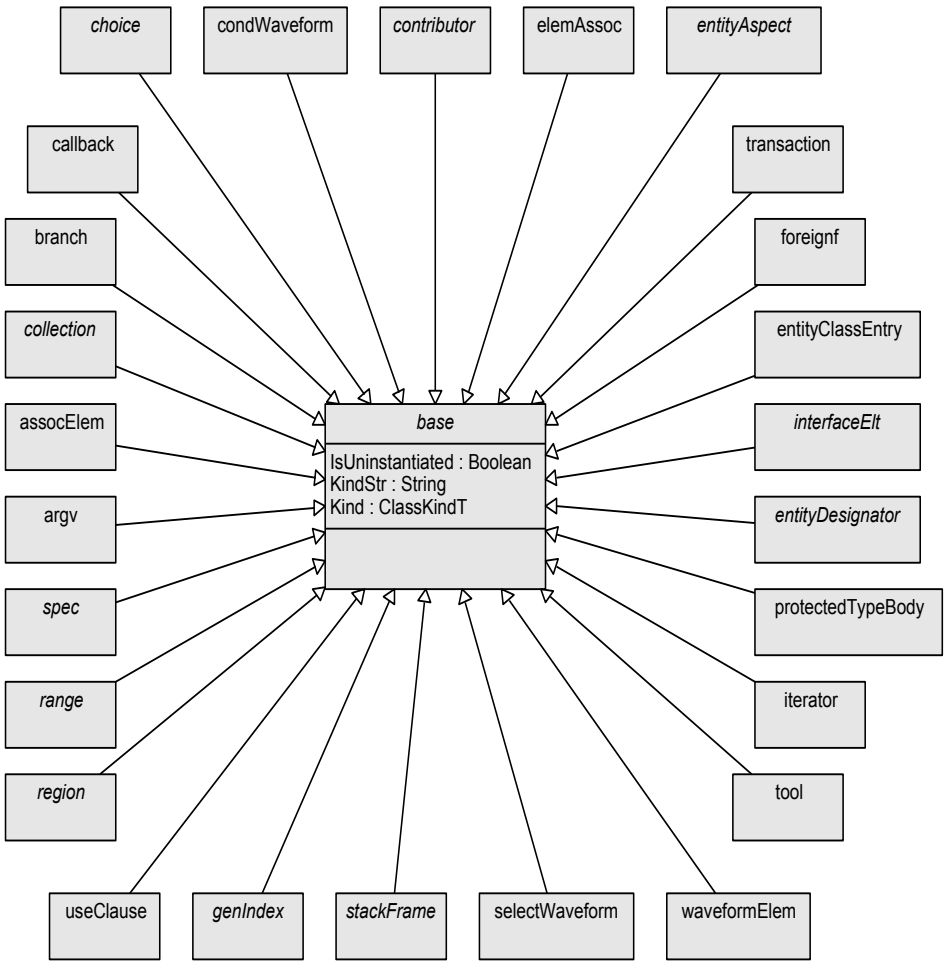

Figure 53—Base class diagram

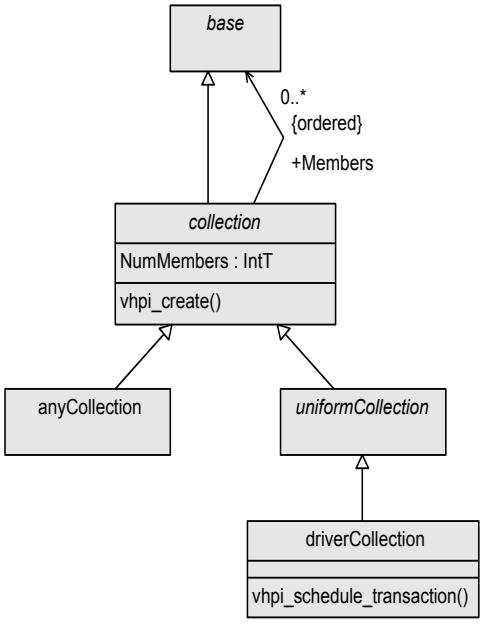

Figure 54—Collection class diagram

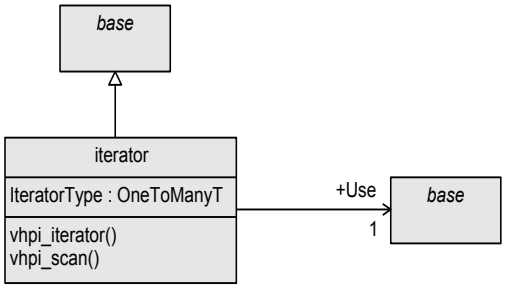

Figure 55—Iterator class diagram

19.17 The stdTool package

The `stdTool` package contains the `Tool` class diagram that specifies aspects of the VHPI information model that relate to the VHPI tool. See Figure 56.

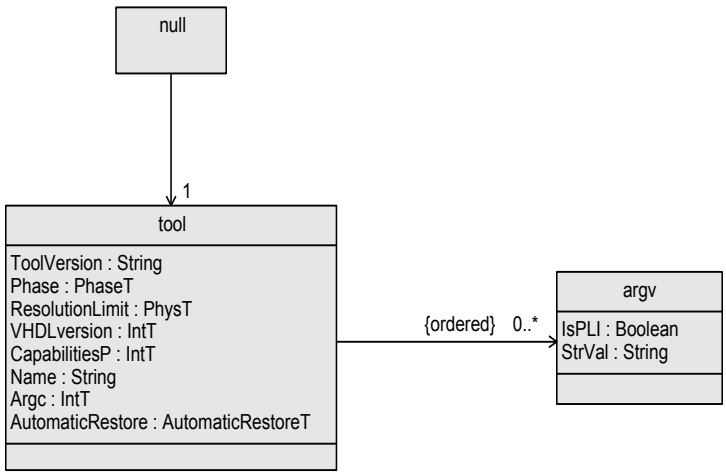

Figure 56—Tool class diagram

19.18 Application contexts

Objects of certain classes in the information model exist during different phases of tool execution. The *application context* of a class specifies whether objects of the class may exist in either or both of the library information model or the design hierarchy information model, and as a consequence, when the object is accessible to VHPI programs. The documentation for each class in the machine-readable information model describes the application context for that class.

Objects in the library information model representing a design unit are created during the analysis phase of tool execution in which the design unit is analyzed. Objects representing previously analyzed design units are accessible from the start of tool execution and remain accessible until the end of tool execution. If a VHPI tool performs the analysis phase, objects representing a design unit being analyzed by the tool are accessible at the end of the analysis phase.

Objects in the design hierarchy information model are created during the elaboration phase of tool execution and are accessible at the end of the elaboration phase. It is an error if a VHPI program accesses objects in the design hierarchy information model during the elaboration phase other than from an elaboration function as specified in 20.4.1.

NOTE—For objects in the library information model, the target objects of associations are also in the library information model. The library information model includes no associations with objects in the design hierarchy information model. For objects in the design hierarchy information model, the target objects of associations are also in the design hierarchy information model, except where specified in the documentation for the association. Those associations from objects in the design hierarchy information model to objects in the library information model allow a VHPI program to navigate between the two information models.

20. VHPI tool execution

20.1 General

This clause describes the way in which foreign models and applications interact with a VHPI tool and the way in which the tool executes VHDL and foreign models. A *foreign model* is a design entity whose architecture is decorated with the 'FOREIGN attribute in the form described in this clause, or a subprogram similarly decorated. A *foreign application* is a VHPI program that does not correspond to design entities or subprograms declared in the VHDL model.

The VHPI supports various execution phases of a VHDL tool. Each phase is identified by a value of the enumeration type `vhpiPhaseT` (see Annex B). A VHPI program determines the current phase of the VHDL tool by calling the VHPI routine `vhpi_get` (see 23.10) supplying the value `vhpiPhaseP` as the first parameter and NULL as the second parameter. The return value of `vhpi_get` is one of the values of `vhpiPhaseT`.

In temporal order, the VHDL tool execution phases are:

a) `vhpiRegistrationPhase`: Indicates the tool has begun executing
b) `vhpiAnalysisPhase`: The analysis of a design file is occurring
c) `vhpiElaborationPhase`: The static elaboration of a design hierarchy is occurring
d) `vhpiInitializationPhase`: The initialization of an elaborated design hierarchy is occurring
e) `vhpiSimulationPhase`: The execution of an elaborated and initialized design hierarchy is occurring
f) `vhpiSavePhase`: The current state of a VHDL model is being saved for possible restart
g) `vhpiRestartPhase`: A previously saved VHDL model is being restarted from the point of its save
h) `vhpiResetPhase`: A VHDL model is being restarted from the state it was in at the end of initialization
i) `vhpiTerminationPhase`: The tool is terminating

NOTE—If a tool does not support a given phase and a VHPI program attempts to register a callback with the callback reason being the start or end of the phase, the `vhpi_register_cb` function raises an error indicating that the callback reason is not implemented.

20.2 Registration phase

20.2.1 General

The registration phase involves the following steps:

a) Foreign models, applications, and libraries of foreign models are registered
b) Each registered and enabled `vhpiCbStartOfTool` callback is executed

The registration phase is complete when all registered and enabled `vhpiCbStartOfTool` callbacks have returned to the VHDL tool. During the registration phase, a call to `vhpi_get(vhpiPhaseP, NULL)` returns `vhpiRegistrationPhase`.

Before a VHPI program can gain access to the internals of a VHDL tool, the program shall register itself with the tool. Through either of two registration mechanisms described in 20.2.2 and 20.2.3, or through decoration of a foreign model with the 'FOREIGN attribute in the form of a standard direct binding (see 20.2.4.3), the tool is supplied with the identity of one or more elaboration, execution, or registration functions in a VHPI program. These functions shall be provided to the tool as entry points in one or more

object libraries. The format of the object libraries and whether the object libraries are statically or dynamically bound to the tool are not specified by this standard. Each registration function shall be of the type `vhpiRegistrationFctT` defined in Annex B.

Prior to the start of processing of any VHDL model by the tool, all registration functions of registered libraries of foreign models and registration functions of selected registered foreign applications are invoked. The manner in which registered foreign applications are selected is not defined by this standard. All such calls to the registration functions shall terminate prior to the tool continuing its execution.

During the registration phase, the only parts of the information model defined by this standard that are available are the objects of the `tool` and `argv` classes. It is an error if a registration function attempts to access other parts of the information model during the registration phase.

A registration function may register callbacks. It is not possible for any VHPI callbacks (see Clause 21) to occur prior to the completion of execution of all registration functions; in particular, registration shall be complete before the `vhpiCbStartOfTool` callback (see 21.3.7.2) can occur.

A tool shall bind an elaboration, execution, or registration function prior to acquiring a pointer to the function or calling the function. A tool is not required to bind such a function immediately upon registration. It is an error if the tool cannot locate an entry point denoted by an elaboration or execution or registration function name.

It is an error if a given foreign model, identified by a unique combination of object library name and model name, is registered more than once by any of the mechanisms defined in this standard.

A foreign application may be registered multiple times with different registration functions. It is an error if a given foreign application, identified by a unique combination of object library name and application name, is registered more than once with the same registration function name by any of the mechanisms defined in this standard.

A library of foreign models may be registered multiple times with different registration functions. It is an error if a given library of foreign models, identified by an object library name, is registered more than once with the same registration function name by any of the mechanisms defined in this standard.

The registration of a VHPI program with a given invocation of a tool does not persist beyond termination of that invocation of the tool.

NOTE 1—A foreign model for which there is no corresponding VHDL architecture or subprogram decorated with the 'FOREIGN attribute may be registered. However, it will have no effect on the design since neither its elaboration function (for a foreign architecture) nor its execution function can be invoked.

NOTE 2—The registration functions are the only entry points in an object library for a foreign application or library of foreign models that need to be externally visible. Entry points for local elaboration and execution functions can be made known to the tool as a consequence of resolving symbols referenced by the registration functions.

20.2.2 Registration using a tabular registry

A tabular registry is a text file containing the registration information for foreign models and applications. Any number of registry files can be passed to a VHDL tool; the mechanism for identifying the files to be passed to a tool is not specified by this standard.

Each entry in the file defines the registration of one foreign model or application, or one library of foreign models. Each entry occupies one line of the file and is a sequence of identifiers separated by one or more space (SPACE or NBSP) characters. Blank lines, containing either no characters or only space characters, and comments may also appear in the file. Space characters preceding an entry in the file or a comment are

ignored. Space characters following an entry in the file are ignored. A comment begins with the characters "--" and continues to the end of the line containing the beginning of the comment.

tabular_registry_file ::= { tabular_registry_entry }

tabular_registry_entry ::=
    foreign_architecture_registry
  | foreign_subprogram_registry
  | foreign_application_registry
  | library_registry

foreign_architecture_registry ::=
  object_library_name model_name **vhpiArchF** elaboration_specifier execution_function_name

foreign_subprogram_registry ::=
    object_library_name model_name **vhpiFuncF null** execution_specifier
  | object_library_name model_name **vhpiProcF null** execution_specifier

foreign_application_registry ::=
  object_library_name application_name **vhpiAppF** registration_function_name **null**

library_registry ::=
  object_library_name **null vhpiLibF** registration_function_name **null**

object_library_name ::= C_identifier | extended_identifier

model_name ::= C_identifier | extended_identifier

application_name ::= C_identifier | extended_identifier

elaboration_specifier ::= elaboration_function_name | **null**

elaboration_function_name ::= C_identifier

execution_specifier ::= execution_function_name | **null**

execution_function_name ::= C_identifier

registration_function_name ::= C_identifier

An object library name denotes a logical name for an object library containing one or more entry points for elaboration, execution, or registration functions. An object library name may or may not be case sensitive, depending on the host environment. The mapping between an object library logical name and a host physical object library is not defined by this standard. It is an error if the host system cannot locate the physical object library identified by an object library name.

A model name is an identifier that, jointly with the object library name, shall uniquely identify a foreign model. An application name is an identifier that, jointly with the object library name, shall uniquely identify a foreign application.

An elaboration function name, execution function name, or registration function name denotes an entry point in the library denoted by the immediately preceding object library name. An elaboration specifier of **null** indicates that no elaboration function is required for the foreign model. An execution specifier of **null**

in a foreign subprogram registry is equivalent to an execution function name that is the same as the immediately preceding model name.

A C identifier is formed from a contiguous sequence of graphical characters according to the rules for forming identifiers in ISO/IEC 9899:1999/Cor 1:2001. The reserved words in a tabular registry entry, **vhpiArchF**, **vhpiFuncF**, **vhpiProcF**, **vhpiAppF**, **vhpiLibF**, and **null**, are case sensitive and shall be written using the combination of uppercase and lowercase letters shown in this standard.

For each entry in the file, the foreign model, foreign application, or library of foreign models whose registration is defined by the entry is registered with the tool reading the tabular registry.

NOTE 1—This standard does not define a default name or location for any tabular registry file.

NOTE 2—A model name or application name alone is not sufficient to uniquely identify a model or application. Different models or applications may have the same model or application names, provided they can be distinguished by different object library names.

NOTE 3—A C identifier that denotes a C function name is the same as the name of the C function defined in the C source code. If an implementation modifies such a name during machine code generation, for example, by prefixing it with an underline character, such modification is not reflected in the use of the name in a tabular registry entry.

Examples:

An example tabular registry:

```
-- registration of a foreign architecture:
myLib orgate vhpiArchF elab_or_gate init_or_gate

-- registration of a foreign function:
myLib myfunc vhpiFuncF null sim_myfunc

-- registration of a foreign application:
myApp appl vhpiAppF register_myapp null

-- registration of a library of models:
myLib null vhpiLibF register_lib null
```

An example registration function for the preceding table:

```
void register_lib() {
  for each model in the library
    vhpi_register_foreignf(...);
}
```

20.2.3 Registration using registration functions

A VHPI program can register a foreign model or application using the `vhpi_register_foreignf` function (see 23.30). The function shall be called during the registration phase of tool execution directly or indirectly from a registration function of a previously registered foreign application or library of foreign models.

20.2.4 Foreign attribute for foreign models

20.2.4.1 General

The value of the 'FOREIGN attribute defined in package STANDARD decorating an architecture or a subprogram may be a string of the form described in this subclause (20.2.4). The value of the attribute is used to identify the VHPI program that implements the foreign model.

The value of the 'FOREIGN attribute for a foreign model is a sequence of identifiers separated by one or more space (SPACE or NBSP) characters. Space characters, if any, preceding or following the sequence of identifiers are ignored.

foreign_attribute_value ::=
    standard_indirect_binding | standard_direct_binding

NOTE 1—The expression in an attribute specification for the 'FOREIGN attribute is required to be locally static (see 7.2). Nonetheless, analysis of a design unit containing a 'FOREIGN attribute specification does not require interpretation of the value of the attribute at the time of analysis.

NOTE 2—An implementation may, as part of elaboration of a 'FOREIGN attribute specification whose value is of the form described in this subclause (20.2.4), perform certain checks, for example, that the C library exists or that the foreign model implementation functions exists.

NOTE 3—The object library name for a foreign model need not be the same as the logical name of the VHDL library containing the architecture or subprogram decorated with the 'FOREIGN attribute.

20.2.4.2 Standard indirect binding

standard_indirect_binding ::=
    VHPI object_library_name model_name

The object library name and model name are described in 20.2.2. The reserved word **VHPI** in a standard indirect binding is case sensitive and shall be written using uppercase letters.

A foreign attribute value in the form of a standard indirect binding specifies sufficient information for the tool to register a foreign model, but not to identify elaboration or execution functions for the foreign model. Identification of functions shall be specified separately using one of the mechanisms described in 20.2.2 or 20.2.3. A VHDL design entity or subprogram decorated with the 'FOREIGN attribute in the form of a standard indirect binding is implemented by the elaboration and execution functions, as appropriate, identified using the same object library name and model name as those that occur in the attribute value.

It is an error if, upon completion of registration, no execution function is specified corresponding to a foreign model for which standard indirect binding is specified in the value of a 'FOREIGN attribute.

NOTE—It is permissible for no elaboration function to be specified corresponding to a foreign architecture for which standard indirect binding is specified.

Example:

The following are analyzed into library `foreignmodels`:

```
package PACKSHELL is
   component C_AND
      port (P1, P2: in BIT; P3: out: BIT);
   end component;

   procedure MYPROC (signal F1: out BIT; constant F2: in INTEGER);
```

```
   attribute FOREIGN of MYPROC: procedure is "VHPI foreignC myCproc";

   function MYFUNC (signal F1: in BIT) return INTEGER;
   attribute FOREIGN of MYFUNC: function is "VHPI foreignC myCfunc";
end package PACKSHELL;

entity C_AND is
   port (P1, P2: in BIT; P3: out: bit);
end C_AND;

architecture MY_C_GATE of C_AND is
   attribute FOREIGN of MY_C_GATE: architecture is
            "VHPI foreignC myCarch";
begin
end architecture MY_C_GATE;
```

The following refer to declarations in the `foreignmodels` library:

```
library FOREIGNMODELS;
use FOREIGNMODELS.PACKSHELL.all;
entity TOP is
end TOP;

architecture MY_VHDL of TOP is
   constant VAL: INTEGER:= 0;
   signal S1, S2, S3: BIT;
begin
   U1: C_AND (S1, S2, S3);
   MYPROC (S1, VAL);

   process (S1)
      variable VA: INTEGER := VAL;
   begin
      VA := MYFUNC (S1);
   end process;
end MY_VHDL;
```

20.2.4.3 Standard direct binding

standard_direct_binding ::=
    standard_direct_architecture_binding | standard_direct_subprogram_binding

standard_direct_architecture_binding ::=
    VHPIDIRECT object_library_specifier elaboration_specifier execution_function_name

standard_direct_subprogram_binding ::=
    VHPIDIRECT object_library_specifier execution_specifier

object_library_specifier ::= object_library_path | **null**

object_library_path ::=
    graphic_character { graphic_character }

A foreign attribute value in the form of a standard direct binding specifies sufficient information for the tool to register a foreign model and to identify elaboration and execution functions, as required, for the foreign model. If the foreign model is a design entity, the standard direct binding shall take the form of a standard direct architecture binding; otherwise, the standard direct binding shall take the form of a standard direct subprogram binding.

An object library specifier denotes a physical name for an object library containing one or more entry points for elaboration or execution functions.

An object library path may or may not be case sensitive, depending on the host environment. If a space character (SPACE or NBSP) is to be used as one of the graphic characters of an object library path, it shall be preceded by a backslash character (the combination of the backslash and space character counting as just the space character). If a backslash is to be used as one of the graphic characters of an extended literal, it shall be doubled (a doubled backslash counting as just one backslash). A host system interprets an object library path in a manner not defined by this standard to locate a physical object library. It is an error if the host system cannot locate the physical object library identified by an object library path.

An object library specifier of **null** indicates that a physical object library is to be determined in an implementation defined manner. It is an error if an object library specifier of **null** is used and the host system cannot locate the physical object library.

The reserved words **VHPIDIRECT** and **null** in a standard direct binding are case sensitive and shall be written using uppercase and lowercase letters, respectively, as shown in this standard.

The elaboration specifier, execution specifier, and execution function name are described in 20.2.2. An execution specifier of **null** in a standard direct subprogram binding is equivalent to an execution function name that is the same as the designator of the subprogram decorated with the foreign attribute value using the same combination of uppercase and lowercase letters that occur in the subprogram declaration for the subprogram, if present, or the subprogram body otherwise.

NOTE—A host system may interpret an object library path by appending an implementation-dependent file-name extension, such as ".so" or ".dll," to derive a file pathname. It is recommended that a file-name extension in an object library path be omitted so that an implementation can append an extension that is appropriate for the host environment.

20.3 Analysis phase

The analysis phase involves the following steps:

a) Each registered and enabled `vhpiCbStartOfAnalysis` callback is executed.

b) One or more design files are analyzed. The manner in which the design files to be analyzed are specified to the tool is not specified by this standard.

c) Each registered and enabled `vhpiCbEndOfAnalysis` callback is executed.

During the analysis phase, a call to `vhpi_get(vhpiPhaseP, NULL)` returns `vhpiAnalysisPhase`.

20.4 Elaboration phase

20.4.1 General

The elaboration phase involves the following steps:

a) Each registered and enabled `vhpiCbStartOfElaboration` callback is executed.

b) The design hierarchy is elaborated, as described in 14.2 through 14.5. This may involve invocation of elaboration functions, if any, for registered foreign architectures.

c) Each registered and enabled `vhpiCbEndOfElaboration` callback is executed.

During the elaboration phase, a call to `vhpi_get(vhpiPhaseP, NULL)` returns `vhpiElaborationPhase`.

An elaboration function shall conform to the rules for a callback function (see Clause 21). It is invoked by the tool in the same way as a `vhpiCbStartOfElaboration` callback. The `reason` member of the callback data structure passed to the elaboration function has the value `vhpiCbStartOfElaboration`. The `obj` member of the callback data structure passed to the elaboration function contains a handle that refers to an object of class `designUnitInst` that represents an instance of the foreign architecture corresponding to the elaboration function. The value of the `user_data` member of the structure is not specified by this standard.

It is an error if an elaboration function accesses the design hierarchy information model other than as follows:

— To access objects navigable from the object of class `designUnitInst`, representing the instance of the foreign architecture body, passed to the elaboration function.

— To use the `vhpi_create` function to create a foreign process, a driver, or a driver collection.

— To use the `vhpi_put_value` function to set the initial value of an elaborated signal within the instance of the corresponding foreign architecture or of an elaborated port of mode **out**, **inout**, or **buffer** of the instance of the corresponding foreign architecture.

NOTE—At the time an elaboration function is invoked, the entire design hierarchy might not have been completely elaborated. Thus, objects that ultimately will be accessible by navigating from the object passed to the elaboration function might not yet exist.

20.4.2 Dynamic elaboration

Dynamic elaboration of a foreign subprogram (see 14.6) involves invocation of the execution function of the foreign subprogram. Dynamic elaboration of a foreign subprogram may occur during the elaboration, initialization, or simulation phases of tool execution

An execution function of a foreign subprogram shall conform to the rules for a callback function (see Clause 21). It is invoked by the tool in the same way as a `vhpiCbStartOfSubpCall` callback. The `reason` member of the callback data structure passed to the elaboration function has the value `vhpiCbStartOfSubpCall`. The `obj` member of the callback data structure passed to the elaboration function contains a handle that refers to an object of class `subpCall` that represents an instance of the call to the subprogram corresponding to the execution function. The value of the `user_data` member of the structure is not specified by this standard.

An execution function of a foreign subprogram may obtain handles to objects representing the elaborated formal parameters and their associated actual parameters. Such handles may become invalid upon completion of the subprogram call. A VHPI program that relies upon such a handle remaining valid after the execution function has returned is erroneous.

Parameters of a foreign subprogram implemented by a VHPI execution function are passed either by copy or by references, as described in 4.2.2. An execution function may use the `vhpi_get_value` function to read the value of a formal parameter of mode **in** or **inout**, and may use the `vhpi_put_value` function to write the value of a formal parameter of mode **out** or **inout**. An execution function may use the `vhpi_schedule_transaction` function to schedule a transaction on a driver for a formal signal parameter of mode **out** or **inout**.

It is an error if the execution function for a foreign function does not provide a result for the function call represented by the object referred to by the `obj` member of the callback data structure. The mechanism for the execution to provide the result is described in 22.5.5.

NOTE—An implementation may, in some cases, be able to statically elaborate parts of interface declarations in a concurrent procedure call statement that invokes a foreign subprogram. In such cases, handles to objects representing the elaborated declarations may remain valid between invocations of the subprogram.

20.5 Initialization phase

The initialization phase involves initializing the design hierarchy, as described in 14.7.5.2. This may involve invocation of execution functions for registered foreign architectures. During the initialization phase, a call to `vhpi_get(vhpiPhaseP, NULL)` returns `vhpiInitializationPhase`.

An execution function of a foreign architecture shall conform to the rules for a callback function (see Clause 21). It is invoked by the tool in the same way as a `vhpiCbStartOfInitialization` callback. The `reason` member of the callback data structure passed to the elaboration function has the value `vhpiCbStartOfInitialization`. The `obj` member of the callback data structure passed to the execution function contains a handle that refers to an object of class `compInstStmt` that represents an instance of the foreign architecture corresponding to the execution function. The value of the `user_data` member of the structure is not specified by this standard.

An execution function of a foreign architecture may access any part of the design hierarchy information model.

NOTE—An execution function of a foreign architecture may register callbacks that occur in later phases of tool execution. Memory allocated by the execution function may be referred to in the `user_data` member of callback data structures used to register such callbacks.

20.6 Simulation phase

The simulation phase involves execution of simulation cycles, including execution of registered and enabled `vhpiCbStartOfSimulation` and `vhpiCbEndOfSimulation` callbacks, as described in 14.7.5.3. During the simulation phase, a call to `vhpi_get(vhpiPhaseP, NULL)` returns `vhpiSimulationPhase`.

20.7 Save phase

A tool may allow a user or a VHPI program to request that the current state of a VHDL model be saved for possible restart. The manner by which such a request is made is not specified by this standard. If a VHPI program makes such a request, the tool shall enter the save phase of tool execution either at the end of the initialization phase, if the request was made before the end of the initialization phase, or at the end of the current simulation cycle otherwise.

The save phase involves the following steps:

a) The tool performs some actions, not specified by this standard, to save the current state of the VHDL model, which includes the time of the next simulation cycle, T_n.

b) Each registered and enabled `vhpiCbStartOfSave` callback is executed.

c) Each registered and enabled `vhpiCbEndOfSave` callback is executed.

During the save phase, a call to `vhpi_get(vhpiPhaseP, NULL)` returns `vhpiSavePhase`.

A VHPI program may register vhpiCbStartOfSave and/or vhpiCbEndOfSave callbacks. During execution of such callbacks, the VHPI program may use the vhpi_put_data (see 23.27) function to include data as part of the saved state. The VHPI program may also register vhpiCbStartOfRestart and/or vhpiCbEndOfRestart callbacks. During the save phase, the tool shall save registration of such callbacks and restore the registration in such a manner that the callbacks can be invoked upon a subsequent restart using the saved state.

NOTE 1—A tool may automatically save part or all of the state of a VHPI program. The flag bits of the value of the AutomaticRestore property of the tool class specify the parts of the state that the tool automatically saves. Depending on which flag bits are set, a VHPI program may need to save information about its handles, callbacks, and private data using the vhpi_put_data function.

NOTE 2—A VHPI program that uses vhpi_put_data to save its state should register a vhpiCbStartOfRestart or vhpiCbEndOfRestart callback and write to the user_data member of the callback data structure the value of the identification number used to save state. The callback function, when invoked, should read the identification number from the user_data member of the callback data structure it is passed and use the id value in calls to the vhpi_get_data function to restore the state.

NOTE 3—If a user interrupts the save phase, through some implementation-defined means, the current state of the model might not be correctly saved. It might not be possible to restart execution of the model using the saved state.

20.8 Restart phase

A tool may allow a user or a VHPI program to request that execution of a VHDL model be restarted from a previously saved state. The manner by which such a request is made is not specified by this standard. If a VHPI program makes such a request, the tool shall enter the restart phase of tool execution either at the end of the initialization phase, if the request was made before the end of the initialization phase, or at the end of the current simulation cycle otherwise.

The restart phase involves the following steps:

a) The tool performs some actions, not specified by this standard, to restore the previously saved state of the VHDL model, including the time of the next simulation cycle, T_n. The manner in which the saved state is identified to the tool is not specified by this standard.

b) Each registered and enabled vhpiCbStartOfRestart callback is executed.

c) Each registered and enabled vhpiCbEndOfRestart callback is executed.

During the restart phase, a call to vhpi_get(vhpiPhaseP, NULL) returns vhpiRestartPhase. After completion of the restart phase, the tool enters the simulation phase, commencing with a new simulation cycle.

NOTE 1—A tool may automatically restore part or all of the state of a VHPI program. The flag bits of the value of the AutomaticRestore property of the tool class specify the parts of the state that the tool automatically restores. Depending on which flag bits are set, a VHPI program may need to reacquire handles, reregister callbacks, and restore private data using the vhpi_get_data function.

NOTE 2—Upon entering the simulation phase from the restart phase, the tool does not execute any vhpiCbStartOfSimulation callbacks.

20.9 Reset phase

A tool may allow a user or a VHPI program to request that execution of a VHDL model be reset to the beginning of the initialization phase. The manner by which such a request is made is not specified by this standard. If a VHPI program makes such a request, the tool shall enter the reset phase of tool execution either at the end of the initialization phase, if the request was made before the end of the initialization phase, or at the end of the current simulation cycle otherwise.

The reset phase involves the following steps:

a) Each registered and enabled `vhpiCbStartOfReset` callback is executed.
b) All callbacks except `vhpiCbEndOfReset` callbacks are removed.
c) The projected output waveform of each driver is reset to its initial contents.
d) The current time, T_c, is reset to be 0 ns.
e) Each registered and enabled `vhpiCbEndOfReset` callback is executed.

During the reset phase, a call to `vhpi_get(vhpiPhaseP, NULL)` returns `vhpiResetPhase`. After completion of the reset phase, the tool enters the initialization phase.

A handle, acquired before the reset phase, that refers to a static object, remains valid during and after the reset phase. A handle, acquired before the reset phase, that refers to a dynamic object, may become invalid during or after the reset phase.

NOTE—A VHPI program that allows for reset should register a `vhpiCbStartOfReset` callback that releases resources and saves information about callbacks that are to be reinstated after reset. It should also register a `vhpiCbEndOfReset` callback that reregisters the callbacks that are to be reinstated.

20.10 Termination phase

The termination phase involves executing each registered and enabled `vhpiCbEndOfTool` callback. When all such callbacks have returned to the tool, the tool may terminate. No further VHPI operations may be called. During the termination phase, a call to `vhpi_get(vhpiPhaseP, NULL)` returns `vhpiTerminationPhase`.

21. VHPI callbacks

21.1 General

A *callback* is a mechanism for a VHPI program to gain control during tool execution. A VHPI program *registers a callback*, providing to the tool a reference to a *callback function* and a *callback reason*, that is, a specification of an event or events that may *trigger* execution of the callback function by the tool. For some callbacks, the trigger event is associated with one or more objects in the information model; such an object is called a *trigger object* of the callback. A foreign model typically registers callbacks during execution of its elaboration or initialization functions, and a foreign application typically registers callbacks during execution of its registration function. A callback function may register subsequent callbacks. As part of registration of a callback, a VHPI program may provide data to be supplied to the callback function when it is invoked.

Depending on the callback reason, a callback is either a *one-time* callback, meaning that the callback function is triggered at most once, or a *repetitive* callback, meaning that the callback function may be triggered multiple times. A callback is in one of three states:

— *enabled*, meaning that the callback function will be called if the trigger event occurs,

— *disabled*, meaning that the callback function will not be called if the trigger event occurs, or

— *mature*, meaning that the callback is a one-time callback whose trigger event has occurred.

If the trigger event of an enabled callback occurs, the callback state is changed to mature if the callback is a one-time callback or remains enabled if the callback is a repetitive callback. In either case, the callback function is then triggered. A VHPI program may register a callback in the enabled state and may disable an enabled callback.

If the trigger event of a disabled callback occurs, the callback state is changed to mature if the callback is a one-time callback or remains disabled if the callback is a repetitive callback. In either case, the callback function is not triggered. A VHPI program may register a callback in the disabled state and may enable a disabled callback. Disabling a callback does not affect the specification of the trigger event of the callback.

A mature callback is not triggered by occurrence of its trigger event subsequent to the occurrence that caused the callback to become mature. Furthermore, the state of a mature callback cannot be changed. A repetitive callback never becomes mature.

NOTE—Disabling a callback simply determines whether or not the callback will be triggered when its trigger event occurs. For example, disabling a callback that is registered to trigger after a given delay and subsequently enabling before expiry of the delay does not postpone the time at which the trigger event occurs.

21.2 Callback functions

21.2.1 General

A callback is represented in the VHPI information model by an object of class `callback`. A VHPI program can obtain a handle to a callback object by navigating the information model. The VHPI provides functions to register, remove, enable, and disable callbacks and to access information about callbacks.

21.2.2 Registering callbacks

A VHPI program may register a callback using the `vhpi_register_cb` function (see 23.9). Prior to calling the function, the VHPI program shall allocate memory for a *callback data structure* of type `vhpiCbDataT` (see Annex B) and write to it values that specify the callback. After the

`vhpi_register_cb` function returns, the tool does not retain any references to the callback data structure or to storage pointed to by members of the callback data structure.

The `reason` member of a callback data structure specifies the callback reason (see 21.3). The `cb_rtn` member shall be a pointer to the callback function. The VHPI program may write a value to the `user_data` member to be passed to the callback function when it is triggered. The value may be of any type, provided it can be cast to a type that is compatible with the type of the `user_data` member. The value is not used by the tool other than being stored so that it can be passed to the callback function. Each of the remaining members either specifies further information, if required for the given callback reason, or is ignored.

NOTE—Since the tool retains no references to the callback data structure provided by a VHPI program to register a callback, the VHPI program may reuse the same data structure to register further callbacks.

21.2.3 Enabling and disabling callbacks

A VHPI program may enable a callback using the `vhpi_enable_cb` function (see 23.8) and may disable a callback using the `vhpi_disable_cb` function (see 23.7).

21.2.4 Removing callbacks

A VHPI program may remove a callback using the `vhpi_remove_cb` function (see 23.32). Once removed, the callback is no longer registered, and occurrence of the callback reason for which the callback was registered does not trigger the callback function. The object representing the callback is removed from the information model. Any handle that refers to the object representing the removed callback is made invalid.

NOTE—Releasing a handle that refers to a callback object neither removes the callback nor changes its state. A handle to the callback can subsequently be acquired by navigating the information model.

21.2.5 Callback information

A VHPI program may obtain information about a registered callback using the `vhpi_get_cb_info` function (see 23.11).

21.2.6 Execution of callbacks

A callback function shall have a single argument that is a constant pointer to a callback data structure and shall have a `void*` return type. When the tool triggers a callback, the tool passes a callback data structure in which

- The value of the `reason` member is the enumeration constant that identifies the callback reason for which the callback was triggered (see 21.3).
- The value of the `cb_rtn` member is a pointer to the callback function.
- The value of the `user_data` member is the value that was provided in the `user_data` member of the callback data structure specified during registration of the callback.

The values of the remaining members depend on the callback reason (see 21.3). The callback data structure passed to a callback function, any time and value structures pointed to by members of the callback data structure, and any buffers for values pointed to by members of the value structure are allocated by the tool.

A callback function that modifies the callback data structure passed to it by the tool is erroneous. If the tool provides a handle to an object in the `obj` member of a callback data structure passed to a callback function, the tool may release the handle upon return of the callback function to the tool. A callback function that releases such a handle is erroneous.

NOTE—Any actions performed by a callback function are subject to rules specified for the step of the simulation cycle in which the callback function is invoked (see 14.7.5) and rules specified for the callback reason for which the callback was triggered (see 21.3).

21.3 Callback reasons

21.3.1 General

This subclause (21.3) describes the callback reasons. Callback reasons are identified by enumeration constants, defined in the VHPI include file, whose names start with the characters `vhpiCb`. In this standard, the term *callback* qualified with an enumeration constant identifying a callback reason refers to a callback that is registered with the reason identified by the enumeration constant. This subclause (21.3) specifies the values required, if any, in the `obj`, `time`, and `value` members of the callback data structure provided by a VHPI program upon registration of a callback for each reason.

If a VHPI program provides a pointer to a time structure in the `time` member of a callback data structure, the VHPI program shall allocate the memory for the time structure.

Similarly, if a VHPI program provides a pointer to a value structure in the `value` member of a callback data structure, the VHPI program shall allocate the memory for the value structure. The value structure shall have the `format` member set to a value of type `vhpiFormatT` specifying the format of a value to be provided to the callback function.

21.3.2 Object callbacks

21.3.2.1 General

An object callback is a callback whose trigger event relates to the value of a variable or a signal, represented by a trigger object. An object callback is a repetitive callback.

In the case of the trigger event of an object callback occurring on a trigger object representing a variable,

- If the variable is of a composite type, the trigger event also occurs on each subelement of the trigger object variable.
- If the variable is a subelement or slice of a composite variable, the trigger event also occurs on each composite variable containing the trigger object variable.
- If the variable is a slice of a composite variable, the trigger event also occurs on each overlapping slice of the trigger object variable.

If the VHPI program registering an object callback provides in the `time` member of the callback data structure a value other than `NULL`, the tool, upon triggering the callback function, provides in the `time` member of the callback structure passed to the callback function a pointer to a time structure representing the time at which the trigger event occurred. The tool does not dereference the value provided by the VHPI program in the `time` member of the callback data structure. If the VHPI program provides the value `NULL` in the `time` member of the callback data structure, the value of the `time` member of the callback structure passed to the callback function is `NULL`.

If the VHPI program registering an object callback provides in the `value` member of the callback data structure a value other than `NULL`, the value shall be a pointer to a value structure. In that case, the tool, upon triggering the callback function, provides in the `value` member of the callback structure passed to the callback function a pointer to a value structure representing the value of the trigger object resulting from the trigger event. The value is represented in the format (see 22.4) specified by the `format` member of the value structure provided by the VHPI program. The tool ignores other members of the value structure provided by the VHPI program. If the VHPI program provides the value `NULL` in the `value` member of the

callback data structure, the value of the `value` member of the callback structure passed to the callback function is `NULL`.

NOTE—Since the tool ignores members of the value structure other than the `format` member, the VHPI program need not allocate memory to be pointed to by the `value` member of the value structure.

21.3.2.2 vhpiCbValueChange

A VHPI program that registers a `vhpiCbValueChange` callback shall provide in the `obj` member of the callback data structure a handle that refers to a trigger object. When the callback is executed, the value of the `obj` member of the callback data structure passed to the callback function is a handle that refers to the trigger object.

The trigger event for a `vhpiCbValueChange` callback is one of:

- A change of value of a variable represented by a trigger object of class `variable` as a result of execution of a variable assignment statement (see 10.6), update of an actual parameter associated with a formal variable parameter of mode **out** or **inout**, or a call by a VHPI program to the `vhpi_put_value` function to update the variable.
- An event on a signal represented by a trigger object of class `signal` as a result of signal update (see 14.7), unless the signal is a port of mode **out**.
- A change of driving value of a port of mode **out** represented by an object of class `outPort` as a result of a source of the port being active.
- A change of driving value of a driver represented by a trigger object of class `driver` as a result of the driver being active (see 14.7).
- An implementation-defined trigger event, other than a trigger event previously listed, that causes the value of the trigger object to change.

NOTE 1—A change in value of a signal or a port caused by a call to the `vhpi_put_value` function with `mode` value `vhpiDeposit` or `vhpiForce` is not a trigger event for a `vhpiCbValueChange` callback.

NOTE 2—A VHPI program cannot register a `vhpiCbValueChange` for an alias of an object.

NOTE 3—An implementation-defined trigger event for a `vhpiCbValueChange` callback may be an event such as a change caused by a user-interface command.

NOTE 4—An event on a signal may result from assignment to the signal by a VHDL description or from a call by a VHPI program to the `vhpi_put_value` function with `mode` value `vhpiDepositPropagate` or `vhpiForcePropagate` to update the signal. Similarly, a change of driving value of a port of mode **out** may result from assignment to a source by a VHDL description or of a call by a VHPI program to the `vhpi_put_value` function with `mode` value `vhpiDepositPropagate` or `vhpiForcePropagate` to update the port; and a change of driving value of a driver may result from assignment to the driven signal by a VHDL description or from a call by a VHPI program to the `vhpi_put_value` function to update the driver. In each case, the change of value is a single trigger event for the `vhpiCbValueChange` callback.

21.3.2.3 vhpiCbForce

A VHPI program that registers a `vhpiCbForce` callback shall provide in the `obj` member of the callback data structure either a handle that refers to a trigger object or `NULL`. If the VHPI program provides a handle that refers to a trigger object, the `vhpiCbForce` callback is associated with that trigger object. If the VHPI program provides `NULL`, the `vhpiCbForce` callback is associated with all objects for which forcing is permitted as trigger objects. In either case, when the callback is executed, the value of the `obj` member of the callback data structure passed to the callback function is a handle that refers to the trigger object upon which the trigger event occurred.

The trigger event for a `vhpiCbForce` callback is one of:

- Execution, without error, of the `vhpi_put_value` function with a `mode` value of `vhpiForce` or `vhpiForcePropagate` to update the value of the trigger object of the callback.

— Execution of a simple force assignment, conditional force assignment or selected force assignment (see 10.5) in which the target or a subelement of the target is represented by the trigger object of the callback.

— Execution, without error, of an implementation-defined force directive, issued from an interactive user or a command source, applied to the trigger object of the callback.

21.3.2.4 vhpiCbRelease

A VHPI program that registers a `vhpiCbRelease` callback shall provide in the `obj` member of the callback data structure either a handle that refers to a trigger object, or `NULL`. If the VHPI program provides a handle that refers to a trigger object, the `vhpiCbRelease` callback is associated with that trigger object. If the VHPI program provides `NULL`, the `vhpiCbRelease` callback is associated with all objects for which forcing is permitted as trigger objects. In either case, when the callback is executed, the value of the `obj` member of the callback data structure passed to the callback function is a handle that refers to the trigger object upon which the trigger event occurred.

The trigger event for a `vhpiCbRelease` callback is one of:

— Execution, without error, of the `vhpi_put_value` function with a `mode` value of `vhpiRelease` to release forcing of the trigger object of the callback.

— Execution of a simple release assignment (see 10.5.2) in which the target or a subelement of the target is represented by the trigger object of the callback.

— Execution, without error, of an implementation-defined release directive, issued from an interactive user or a command source, applied to the trigger object of the callback.

21.3.2.5 vhpiCbTransaction

A VHPI program that registers a `vhpiCbTransaction` callback shall provide in the `obj` member of the callback data structure a handle that refers to a trigger object of class `driver` or `signal`. When the callback is executed, the value of the `obj` member of the callback data structure passed to the callback function is a handle that refers to the trigger object.

The trigger event for a `vhpiCbTransaction` callback is the trigger object becoming active (see 14.7.3.1).

21.3.3 Foreign model callbacks

21.3.3.1 General

A foreign model callback is a callback that allows a foreign model to achieve an effect similar to that of a wait statement.

21.3.3.2 vhpiCbTimeOut and vhpiCbRepTimeOut

The `vhpiCbTimeOut` callback is a one-time callback, whereas the `vhpiCbRepTimeOut` callback is a repetitive callback.

A VHPI program that registers a `vhpiCbTimeOut` or a `vhpiCbRepTimeOut` callback shall provide in the `time` member of the callback data structure a pointer to a time structure that specifies a *timeout interval*. The trigger event for these callbacks is the expiry of the timeout interval after the callback was registered. In the case of the `vhpiCbRepTimeOut` callback, further trigger events occur upon expiry of successive intervals equal to the timeout interval, for as long as the simulation is not complete. Execution of `vhpiCbTimeOut` and `vhpiCbRepTimeOut` callbacks is described in 14.7.5.3.

The values of the obj and value members of the callback data structure for a vhpiCbTimeOut or vhpiCbRepTimeOut callback are ignored by the tool.

NOTE 1—A foreign model that registers a vhpiCbTimeOut callback is similar in effect to a nonpostponed process that executes a wait statement with a timeout clause. A foreign model can achieve an effect similar to a postponed process executing a wait statement with a timeout clause by registering a vhpiCbTimeOut callback that, in turn, registers a vhpiCbStartOfPostponed callback.

NOTE 2—A vhpiCbTimeOut or a vhpiCbRepTimeOut callback cannot be registered or enabled by a vhpiCbEndOfTimeStep or vhpiCbRepEndOfTimeStep callback function (see 21.3.6.8).

21.3.3.3 vhpiCbSensitivity

The vhpiCbSensitivity callback is a repetitive callback.

A VHPI program that registers a vhpiCbSensitivity callback shall provide in the obj member of the callback data structure handle that refers to either an object of class signal or an object of class anyCollection representing a collection of objects of class signal. In the former case, the trigger event for the callback is an event on the signal represented by the object of class signal. In the latter case, the set of signals represented by the objects of class signal is referred to as the *sensitivity set* of the callback, and the trigger event for the callback is an event on any of the signals in the sensitivity set of the callback.

If the VHPI program registering the callback provides in the time member of the callback data structure a value other than NULL, the tool, upon triggering the callback function, provides in the time member of the callback structure passed to the callback function a pointer to a time structure representing the time at which the trigger event occurred. The tool does not dereference the value provided by the VHPI program in the time member of the callback data structure. If the VHPI program provides the value NULL in the time member of the callback data structure, the value of the time member of the callback structure passed to the callback function is NULL.

If the VHPI program registering the callback provides in the value member of the callback data structure a value other than NULL, one of the following occurs:

— If the VHPI program provides in the obj member of the callback data structure a handle that refers to an object of class signal, the tool ignores the value of the value member of the callback data structure.

— Otherwise, the tool, upon triggering the callback function, provides in the value member of the callback structure passed to the callback function a pointer to *sensitivity-set bitmap*, that is, a value structure indicating on which signals in the sensitivity set of the callback an event occurred. The value structure represents a one-dimensional array of integers using the format vhpiIntVecVal. The number of elements in the array is given by the expression $\lceil |s| / 32 \rceil$, where *s* denotes the sensitivity set. The bits of the elements correspond in an implementation-defined manner to the members of the sensitivity set. A bit corresponding to a given signal in the sensitivity set is 1 if there is an event on the given signal, or 0 otherwise. A VHPI program may use the sensitivity-set bitmap macros (see B.2) to determine whether the bit corresponding to a signal is 1 or 0.

The tool does not dereference the value provided by the VHPI program in the value member of the callback data structure.

If the VHPI program provides the value NULL in the value member of the callback data structure, the value of the value member of the callback structure passed to the callback function is NULL.

NOTE—A foreign model that registers a vhpiCbSensitivity callback is similar in effect to a nonpostponed process that executes a wait statement that is sensitive to the signals. A foreign model can achieve an effect similar to a postponed process executing a wait statement that is sensitive to signals by registering a vhpiCbSensitivity

callback that, in turn, registers a `vhpiCbStartOfPostponed` callback. The values of the signals during execution of the latter callback may be different from the values that caused the former callback to trigger.

21.3.4 Statement callbacks

21.3.4.1 General

A statement callback is a callback whose trigger event relates to execution of one or more statements of suspension or resumption of a process. A statement callback is a repetitive callback.

The values of the `time` and `value` members of the callback data structure for a statement callback are ignored by the tool.

21.3.4.2 vhpiCbStmt

A VHPI program that registers a `vhpiCbStmt` callback shall provide in the `obj` member of the callback data structure a handle that refers to a trigger object of class `seqStmt`, `branch`, or `eqProcessStmt` in the design hierarchy information model.

The trigger event for a `vhpiCbStmt` callback is determined as follows:

— If the trigger object is of class `seqStmt` other than an object of class `loopStmt`, the trigger event occurs immediately before execution of the statement represented by the trigger object.

— If the trigger object is of class `loopStmt`, the trigger event occurs immediately before execution of the loop statement represented by the trigger object. Subsequent trigger events occur in each iteration, if any, of the loop statement after the first iteration. In the case of a loop statement without an iteration scheme, subsequent trigger events occur immediately before execution of the sequence of statements enclosed in the loop statement. In the case of a loop statement with a **while** iteration scheme, subsequent trigger events occur immediately before evaluation of the condition in the iteration scheme. In the case of a loop statement with a **for** iteration scheme, subsequent trigger events occur immediately before assignment of a value to the loop parameter.

— If the trigger object is of class `branch` and is associated with an object of class `ifStmt`, the trigger event occurs immediately before evaluation of the condition represented by the trigger object.

— If the trigger object is of class `branch` and is associated with an object of class `caseStmt`, the trigger event occurs immediately before execution of the sequence of statements in the case statement alternative represented by the trigger object.

— If the trigger object is of class `processStmt`, the trigger event occurs immediately before execution of the first statement in the statement part of the process statement represented by the trigger object.

— If the trigger object is of class `concProcCallStmt`, the trigger event occurs immediately before execution of the first statement in the statement part of the subprogram body called by the concurrent procedure call statement represented by the trigger object.

— If the trigger object is of class `eqProcessStmt` and not of class `processStmt` or `concProcCallStmt`, the trigger event occurs at the same time as a trigger event would occur immediately before execution of the first statement in the equivalent process statement of the statement represented by the trigger object, were the equivalent process statement executed instead of the statement represented by the trigger object.

If the trigger object is of class `processStmt` or `concProcCallStmt`, when the callback is executed, the value of the `obj` member of the callback data structure passed to the callback function is a handle that refers to an object representing the first statement in the statement part of the called subprogram body or process statement, respectively. Otherwise, the value of the `obj` member of the callback data structure passed to the callback function is a handle that refers to the trigger object.

A tool may perform optimizations that render an object representing a statement inaccessible. It is an error if the handle in the `obj` member of the registration callback data structure refers to such an object. A tool may perform optimizations that alter the order of execution of statements in a statement part. A VHPI program that depends on the order of execution of `vhpiStmt` callbacks associated with statements whose execution order is so altered is erroneous.

21.3.4.3 vhpiCbResume

A VHPI program that registers a `vhpiCbResume` callback shall provide in the `obj` member of the callback data structure a handle that refers to a trigger object of class `eqProcessStmt` in the design hierarchy information model. A `vhpiCbResume` callback is triggered as described in 14.7.5.3.

If the process represented by the trigger object was suspended as a result of executing an explicit wait statement, or the process represented by the trigger object is a process statement in which a sensitivity list appears after the reserved word **process**, when the callback is executed, the value of the `obj` member of the callback data structure passed to the callback function is a handle that refers to an object representing the sequential statement to be executed upon resumption of the process. Otherwise, the value of the `obj` member of the callback data structure passed to the callback function is a handle that refers to the trigger object.

An implementation may optimize execution of a wait statement in such a way as to obviate some or all resumptions and repeated suspensions of a process provided that, when an event occurs on any signal in the sensitivity set and that event would result in the condition evaluating to TRUE, the process does resume. A VHPI program that depends on triggering of `vhpiCbResume` callbacks for resumptions so obviated is erroneous.

21.3.4.4 vhpiCbSuspend

A VHPI program that registers a `vhpiCbSuspend` callback shall provide in the `obj` member of the callback data structure a handle that refers to a trigger object of class `eqProcessStmt` in the design hierarchy information model. A `vhpiCbSuspend` callback is triggered as described in 14.7.5.2 and 14.7.5.3.

If the process represented by the trigger object is suspended as a result of executing an explicit wait statement, when the callback is executed, the value of the `obj` member of the callback data structure passed to the callback function is a handle that refers to an object representing the wait statement. Otherwise, the value of the `obj` member of the callback data structure passed to the callback function is a handle that refers to the trigger object.

An implementation may optimize execution of a wait statement in such a way as to obviate some or all resumptions and repeated suspensions of a process provided that, when an event occurs on any signal in the sensitivity set and that event would result in the condition evaluating to TRUE, the process does resume. A VHPI program that depends on triggering of `vhpiCbSuspend` callbacks for repeated suspensions so obviated is erroneous.

21.3.4.5 vhpiCbStartOfSubpCall

A VHPI program that registers a `vhpiCbStartOfSubpCall` callback shall provide in the `obj` member of the callback data structure a handle that refers to a trigger object of class `subpCall` in the design hierarchy information model, or to a trigger object of class `subpDecl` in the library information model. The trigger event of a `vhpiStartOfSubpCall` callback occurs immediately after elaboration of the interface list of the called subprogram and association of actual parameters with the formal parameters (see 14.6). If the trigger object is of class `subpDecl`, the trigger event occurs for any call to the subprogram represented by the trigger object.

When the callback is executed, if the trigger object is of class `subpCall`, the value of the `obj` member of the callback data structure passed to the callback function is a handle that refers to the trigger object. Otherwise, the value of the `obj` member is a handle to an object of class `subpCall` that represents the call to the subprogram represented by the trigger object.

NOTE—A VHPI program may obtain a handle to a subprogram call by iterating on the sequential statements of a region, such as a process or a dynamically elaborated instance of a subprogram.

21.3.4.6 vhpiCbEndOfSubpCall

A VHPI program that registers a `vhpiCbEndOfSubpCall` callback shall provide in the `obj` member of the callback data structure a handle that refers to a trigger object of class `subpCall` in the design hierarchy information model. The trigger event of a `vhpiEndOfSubpCall` callback occurs immediately after completion of execution of the statements of the called subprogram. In the case of the called subprogram being a function, the trigger event occurs before the result of the function is used as the value of the function call. In the case of the called subprogram being a procedure, the trigger event occurs before any formal variable parameters of mode **out** or **inout** passed by copy are copied back into the associated actual parameters.

When the callback is executed, the value of the `obj` member of the callback data structure passed to the callback function is a handle that refers to the trigger object.

NOTE 1—A VHPI program may obtain a handle to a subprogram call by iterating on the sequential statements of a region, such as a process or a dynamically elaborated instance of a subprogram.

NOTE 2—A VHPI program may, in a `vhpiCbEndOfSubpCall` callback, use the `vhpi_put_value` function to update the result of a function or the values of formal variable parameters of mode **out** or **inout**.

21.3.5 Time callbacks

21.3.5.1 General

A time callback is a callback whose trigger event relates to progress of simulation time.

The values of the `obj` and `value` members of the callback data structure for a time callback are ignored by the tool.

21.3.5.2 vhpiCbAfterDelay and vhpiCbRepAfterDelay

The `vhpiCbAfterDelay` callback is a one-time callback, whereas the `vhpiCbRepAfterDelay` callback is a repetitive callback.

A VHPI program that registers a `vhpiCbAfterDelay` or a `vhpiCbRepAfterDelay` callback shall provide in the `time` member of the callback data structure a pointer to a time structure that specifies a *timeout interval*. The trigger event for these callbacks is the expiry of the timeout interval after the callback was registered. In the case of the `vhpiCbRepAfterDelay` callback, further trigger events occur upon expiry of successive intervals equal to the timeout interval, for as long as the simulation is not complete. Execution of `vhpiCbAfterDelay` and `vhpiCbRepAfterDelay` callbacks is described in 14.7.5.3.

NOTE—A `vhpiCbAfterDelay` or a `vhpiCbRepAfterDelay` callback cannot be registered or enabled by a `vhpiCbEndOfTimeStep` or `vhpiCbRepEndOfTimeStep` callback function (see 21.3.6.8).

21.3.6 Simulation phase callbacks

21.3.6.1 General

A simulation phase callback is a callback whose trigger event relates to steps of the simulation cycle (see 14.7.5.3). Simulation phase callbacks identified by enumeration constants whose names include the letters "`Rep`" are repetitive callbacks; the remaining simulation phase callbacks are one-time callbacks.

The values of the `obj`, `time`, and `value` members of the callback data structure for a simulation phase callback are ignored by the tool.

21.3.6.2 vhpiCbNextTimeStep and vhpiCbRepNextTimeStep

The trigger event for a `vhpiCbNextTimeStep` or `vhpiCbRepNextTimeStep` callback is occurrence of substep 1) of step b) of a simulation cycle that is not a delta cycle (see 14.7.5.3).

21.3.6.3 vhpiCbStartOfNextCycle and vhpiCbRepStartOfNextCycle

The trigger event for a `vhpiCbStartOfNextCycle` or `vhpiCbRepStartOfNextCycle` callback is occurrence of step b) of the initialization phase (see 14.7.5.2) or occurrence of substep 2) of step b) of a simulation cycle (see 14.7.5.3).

21.3.6.4 vhpiCbStartOfProcesses and vhpiCbRepStartOfProcesses

The trigger event for a `vhpiCbStartOfProcesses` or `vhpiCbRepStartOfProcesses` callback is occurrence of step e) of the initialization phase (see 14.7.5.2) or occurrence of substep 1) of step f) of a simulation cycle (see 14.7.5.3).

21.3.6.5 vhpiCbEndOfProcesses and vhpiCbRepEndOfProcesses

The trigger event for a `vhpiCbEndOfProcesses` or `vhpiCbRepEndOfProcesses` callback is occurrence of step h) of the initialization phase (see 14.7.5.2) or occurrence of substep 4) of step f) of a simulation cycle (see 14.7.5.3).

21.3.6.6 vhpiCbLastKnownDeltaCycle and vhpiCbRepLastKnownDeltaCycle

The trigger event for a `vhpiCbLastKnownDeltaCycle` or `vhpiCbLastKnownDeltaCycle` callback is occurrence of substep 1) of step h) of a simulation cycle (see 14.7.5.3).

21.3.6.7 vhpiCbStartOfPostponed and vhpiCbRepStartOfPostponed

The trigger event for a `vhpiCbStartOfPostponed` or `vhpiCbRepStartOfPostponed` callback is occurrence of step i) of the initialization phase (see 14.7.5.2) or occurrence of substep 3) of step h) of a simulation cycle (see 14.7.5.3).

21.3.6.8 vhpiCbEndOfTimeStep and vhpiCbRepEndOfTimeStep

The trigger event for a `vhpiCbEndOfTimeStep` or `vhpiCbRepEndOfTimeStep` callback is occurrence of substep 6) of step h) of a simulation cycle (see 14.7.5.3).

It is an error if a `vhpiCbEndOfTimeStep` or `vhpiCbRepEndOfTimeStep` callback causes activity on a driver or a signal (see 14.7.3.1) or registers or enables a `vhpiCbAfterDelay`, `vhpiCbRepAfterDelay`, `vhpiCbTimeout`, or `vhpiCbRepTimeOut` callback.

NOTE—The restrictions on scheduling activity and registering future callbacks prevent a `vhpiCbEndOfTimeStep` or `vhpiCbRepEndOfTimeStep` callback from affecting the time of the next simulation cycle (see 14.7.5.1). A callback can cause activity using the `vhpi_schedule_transaction` function or the `vhpi_put_value` function with a `mode` value of `vhpiDepositPropagate` or `vhpiForcePropagate`. A `vhpiCbEndOfTimeStep` or `vhpiCbRepEndOfTimeStep` callback cannot legally use those functions for that purpose.

21.3.7 Action callbacks

21.3.7.1 General

An action callback is a callback whose trigger event relates to occurrence of phases of tool execution and other aspects of tool execution. The `vhpiCbQuiescence`, `vhpiEnterInteractive`, `vhpiExitInteractive`, and `vhpiSigInterrupt` callbacks are repetitive callbacks; the remaining action callbacks are one-time callbacks.

The values of the `obj`, `time`, and `value` members of the callback data structure for an action callback are ignored by the tool.

NOTE—A VHPI program may register an action callback whose trigger event cannot subsequently occur. For example, the VHPI program may register an action callback whose trigger event is occurrence of a given phase of tool execution after all occurrences of the phase have completed. Registration of such a callback is not an error.

21.3.7.2 vhpiCbStartOfTool and vhpiCbEndOfTool

The trigger event of a `vhpiCbStartOfTool` callback occurs immediately prior to completion of the `vhpiRegistrationPhase` phase of tool execution (see 20.2). The trigger event of a `vhpiCbEndOfTool` callback occurs during the `vhpiTerminationPhase` phase of tool execution (see 20.10).

21.3.7.3 vhpiCbStartOfAnalysis and vhpiCbEndOfAnalysis

The trigger event of a `vhpiCbStartOfAnalysis` callback occurs upon commencement of the `vhpiAnalysisPhase` phase of tool execution, and the trigger event of a `vhpiCbEndOfAnalysis` callback occurs immediately prior to completion of the `vhpiAnalysisPhase` phase of tool execution (see 20.3).

If a tool supports the `vhpiAnalysisPhase` phase of tool execution, it shall support the `vhpiCbStartOfAnalysis` and `vhpiCbEndOfAnalysis` callbacks.

21.3.7.4 vhpiCbStartOfElaboration and vhpiCbEndOfElaboration

The trigger event of a `vhpiCbStartOfElaboration` callback occurs upon commencement of the `vhpiElaborationPhase` phase of tool execution, and the trigger event of a `vhpiCbEndOfElaboration` callback occurs immediately prior to completion of the `vhpiElaborationPhase` phase of tool execution (see 20.4).

If a tool supports the `vhpiElaborationPhase` phase of tool execution, it shall support the `vhpiCbStartOfElaboration` and `vhpiCbEndOfElaboration` callbacks.

21.3.7.5 vhpiCbStartOfInitialization and vhpiCbEndOfInitialization

The trigger event of a `vhpiCbStartOfInitialization` callback occurs upon commencement of the `vhpiInitializationPhase` phase of tool execution, and the trigger event of a `vhpiCbEndOfInitialization` callback occurs immediately prior to completion of the `vhpiInitializationPhase` phase of tool execution (see 14.7.5.2 and 20.5).

If a tool supports the `vhpiInitializationPhase` phase of tool execution, it shall support the `vhpiCbStartOfInitialization` and `vhpiCbEndOfInitialization` callbacks.

21.3.7.6 vhpiCbStartOfSimulation and vhpiCbEndOfSimulation

The trigger event of a `vhpiCbStartOfSimulation` callback occurs upon commencement of the `vhpiSimulationPhase` phase of tool execution, and the trigger event of a `vhpiCbEndOfSimulation` callback occurs immediately prior to completion of the `vhpiSimulationPhase` phase of tool execution (see 14.7.5.3 and 20.6).

It is an error if a `vhpiCbEndOfSimulation` callback calls the `vhpi_put_value` function or the `vhpi_schedule_transaction` function either to update the projected output waveform of a driver or to update the value of an object.

If a tool supports the `vhpiSimulationPhase` phase of tool execution, it shall support the `vhpiCbStartOfSimulation` and `vhpiCbEndOfSimulation` callbacks.

NOTE—A `vhpiCbEndOfSimulation` callback may access the library and design hierarchy information models and may read the values of objects using the `vhpi_get_value` function. Furthermore, it may request a control action using `vhpi_control`, for example, to reset, restart, or terminate simulation.

21.3.7.7 vhpiCbQuiescense

The trigger event of a `vhpiCbQuiescense` callback occurs during substep 7) of step h) of a simulation cycle as described in 14.7.5.3.

NOTE—A `vhpiCbQuiescense` callback may cause further simulation cycles by updating the projected output waveform of a driver either by calling the `vhpi_schedule_transaction` function or by calling the `vhpi_put_value` transaction with a `mode` value of `vhpiDepositPropagate` or `vhpiForcePropagate`.

21.3.7.8 vhpiCbEnterInteractive

The trigger event of a `vhpiCbEnterInteractive` callback occurs immediately prior to a tool entering a mode of operation in which it accepts directives from an interactive command source. The circumstances under which a tool enters such a mode and the operation of the tool in that mode are implementation-defined.

21.3.7.9 vhpiCbExitInteractive

The trigger event of a `vhpiCbExitInteractive` callback occurs immediately upon to a tool leaving a mode of operation in which it accepts directives from an interactive command source and resuming tool execution as described in Clause 20. The circumstances under which a tool leaves such a mode are implementation-defined.

21.3.7.10 vhpiCbSigInterrupt

The trigger event of a `vhpiCbSigInterrupt` callback occurs in response to an implementation-defined interrupt event.

NOTE—The interrupt event may be an event that occurs asynchronously with respect to tool execution or an exception event.

21.3.8 Save, restart, and reset callbacks

21.3.8.1 General

The trigger events of callbacks described in this subclause (21.3.8) relate to occurrence of the save, restart, and reset phases of tool execution.

The `vhpiCbStartOfRestart` and `vhpiCbEndOfRestart` callbacks are one-time callbacks; the remainder are repetitive callbacks.

The values of the `obj`, `time`, and `value` members of the callback data structure for a save, restart, or reset callback are ignored by the tool.

21.3.8.2 vhpiCbStartOfSave and vhpiCbEndOfSave

The trigger event of a `vhpiCbStartOfSave` or `vhpiCbEndOfSave` callback occurs during the `vhpiSavePhase` phase of tool execution (see 20.7).

If a tool supports the `vhpiSavePhase` phase of tool execution, it shall support the `vhpiCbStartOfSave` and `vhpiCbEndOfSave` callbacks.

NOTE 1—A `vhpiCbStartOfSave` or `vhpiCbEndOfSave` callback that registers a `vhpiCbStartOfRestart` or `vhpiCbEndOfRestart` callback should not use the `user_data` member of the callback data structure to convey a pointer to saved data, since the data may be restored to a different location when the simulation is restarted. Instead, the callback should use the `user_data` member to convey an identification number for data saved using the `vhpi_put_data` and `vhpi_get_data` functions.

NOTE 2—A VHPI program whose `vhpiCbStartOfSave` callback modifies data in preparation for saving (for example, by converting pointers to relocatable addresses) may register a `vhpiCbEndOfSave` callback to reverse the modification to allow continued simulation.

NOTE 3—The order in which `vhpiCbStartOfSave` and `vhpiCbEndOfSave` callbacks are executed is not required to be the same as the order in which the callbacks were registered, except that all enabled `vhpiCbStartOfSave` callbacks are executed before any `vhpiCbEndOfSave` callbacks.

NOTE 4—No callbacks are triggered between completion of all enabled `vhpiCbStartOfSave` callbacks and triggering of any `vhpiCbEndOfSave` callbacks.

NOTE 5—During execution of a `vhpiCbStartOfSave` or `vhpiCbEndOfSave` callback, the current simulation time returned by the `vhpi_get_time` function is either 0 ns, if the save was requested during the initialization phase, or the time of the current simulation cycle, if the save was requested during a simulation cycle.

21.3.8.3 vhpiCbStartOfRestart and vhpiCbEndOfRestart

The trigger event of a `vhpiCbStartOfRestart` or `vhpiCbEndOfRestart` callback occurs during the `vhpiRestartPhase` phase of tool execution (see 20.8).

It is an error if a `vhpiCbStartOfRestart` callback is registered other than during the `vhpiSavePhase` of tool execution.

If a tool supports the `vhpiRestartPhase` phase of tool execution, it shall support the `vhpiCbStartOfRestart` and `vhpiCbEndOfRestart` callbacks.

NOTE 1—A VHPI program whose `vhpiCbStartOfRestart` callback restores data using the `vhpi_get_data` function may register a `vhpiCbEndOfRestart` callback to reinstate callbacks required for continued simulation of the restored model.

NOTE 2—No callbacks are triggered between completion of all enabled `vhpiCbStartOfRestart` callbacks and triggering of any `vhpiCbEndOfRestart` callbacks.

NOTE 3—During execution of a `vhpiCbStartOfRestart` callback, the current simulation time returned by the `vhpi_get_time` function is either 0 ns, if the restart was requested during the initialization phase, or the time of the current simulation cycle, if the restart was requested during a simulation cycle. During execution of a `vhpiCbEndOfRestart` callback, the current simulation time returned by the `vhpi_get_time` function is either 0 ns, if the save of the model restarted was requested during the initialization phase of execution of the restarted model, or the time of the simulation cycle during which the save of the restarted model was requested, if the save was requested during a simulation cycle.

21.3.8.4 vhpiCbStartOfReset and vhpiCbEndOfReset

The trigger event of a `vhpiCbStartOfReset` or `vhpiCbEndOfReset` callback occurs during the `vhpiResetPhase` phase of tool execution (see 20.9).

If a tool supports the `vhpiResetPhase` phase of tool execution, it shall support the `vhpiCbStartOfReset` and `vhpiCbEndOfReset` callbacks.

NOTE 1—A VHPI program whose `vhpiCbStartOfReset` callback resets the state of private data may register a `vhpiCbEndOfReset` callback to reinstate callbacks or register new callbacks required for repeated simulation of the model.

NOTE 2—No callbacks are triggered between completion of all enabled `vhpiCbStartOfReset` callbacks and triggering of any `vhpiCbEndOfReset` callbacks.

NOTE 3—During execution of a `vhpiCbStartOfReset` callback, the current simulation time returned by the `vhpi_get_time` function is either 0 ns, if the reset was requested during the initialization phase, or the time of the current simulation cycle, if the reset was requested during a simulation cycle. During execution of a `vhpiCbEndOf-Reset` callback, the current simulation time returned by the `vhpi_get_time` function is 0 ns.

22. VHPI value access and update

22.1 General

This clause describes the data structures and operations provided in the VHPI for reading and updating values of objects in a VHDL model.

22.2 Value structures and types

22.2.1 General

The VHPI header file (see Annex B) defines a number of data types that are used by VHPI function. They are described in this subclause (22.2).

It is an error if a VHPI program uses a given type described in this clause to represent a VHDL scalar type, and there are position numbers in the scalar type that exceed the range of position numbers that can be represented in the given type.

22.2.2 vhpiEnumT and vhpiSmallEnumT

A value of type `vhpiEnumT` or `vhpiSmallEnumT` represents a value of a VHDL enumeration type. A value of type `vhpiEnumT` shall be represented with at least 32 bits, and a value of type `vhpiSmallEnumT` shall be represented with at least 8 bits. The value represented by a given value of either type is an enumeration value whose position number is the given value, interpreted as an unsigned binary number.

22.2.3 vhpiIntT and vhpiLongIntT

A value of type `vhpiIntT` or `vhpiLongIntT` represents a value of a VHDL integer type. A value of type `vhpiIntT` shall be represented with at least 32 bits, and a value of type `vhpiLongIntT` shall be represented with at least 64 bits. The value represented by a given value of either type is the given value, interpreted as a signed twos-complement binary number.

22.2.4 vhpiCharT

A value of type `vhpiCharT` represents a value of a VHDL character type. The value shall be represented with at least 8 bits. The value represented by a given value of type `vhpiCharT` is a character value whose position number is the given value, interpreted as an unsigned binary number.

22.2.5 vhpiRealT

A value of type `vhpiRealT` represents a value of a VHDL floating-point type. The value shall be represented with at least 64 bits. The value represented by a given value of type `vhpiRealT` is the given value, interpreted according to the chosen representation for floating-point types (see 5.2.5.1).

22.2.6 vhpiPhysT and vhpiSmallPhysT

A value of type `vhpiPhysT` is called a *physical structure* and represents a value of a physical type. The *position number* of a physical structure is the signed integer represented by the concatenation of the `high` and `low` members of the physical structure to form a 64-bit twos-complement binary number, with the `high` member as the most significant part and the `low` member as the least significant part.

A value of type `vhpiSmallPhysT` also represents a value of a physical type. The value shall be represented with at least 32 bits. The *position number* of the value of type `vhpiSmallPhysT` is the value interpreted as a signed twos-complement binary number.

If a physical structure or value of type `vhpiSmallPhysT` occurs as part of a value structure or as an element of an array pointed to by a value structure, its position number determines the value represented by the value structure or value of type `vhpiSmallPhysT`, as described in 22.2.8. Otherwise, the physical structure or value of type `vhpiSmallPhysT` represents a value of a physical type. The value is the product of the position number of the physical structure or value of type `vhpiSmallPhysT` and a unit determined from the context in which the physical structure or value of type `vhpiSmallPhysT` occurs.

22.2.7 vhpiTimeT

A value of type `vhpiTimeT` is called *time structure* and represents a time value. The *position number* of a time structure is the signed integer represented by the concatenation of the `high` and `low` members of the time structure to form a 64-bit twos-complement binary number, with the `high` member as the most significant part and the `low` member as the least significant part.

If a time structure occurs as part of a value structure or as an element of an array pointed to by a value structure, its position number determines the value represented by the value structure, as described in 22.2.8. Otherwise, the time structure represents a value of type TIME defined in package STANDARD. The value is the product of the position number of the time structure and the resolution limit of the tool.

NOTE—A VHPI program can determine the resolution limit with the function call `vhpi_get_phys(vhpiResolutionLimit, NULL)`.

22.2.8 vhpiValueT

A value of type `vhpiValueT` is called a *value structure* and represents a scalar value, a one-dimensional array of scalar values, or a value of any type represented in an implementation-defined internal representation.

The `format` member of a value structure specifies the *format* of the value structure, that is, a value of type `vhpiFormatT` that determines how the value is represented. The `value` member of the value structure is a union that contains the value in the appropriate representation. The following formats are specified by this standard:

`vhpiBinStrVal`	The value structure represents a scalar value. The position number of the scalar value is represented in the `str` member of the `value` member using a pointer to a string of binary digit characters interpreted as a binary number.
`vhpiOctStrVal`	The value structure represents a scalar value. The position number of the scalar value is represented in the `str` member of the `value` member using a pointer to a string of octal digit characters interpreted as an octal number.
`vhpiDecStrVal`	The value structure represents a scalar value. The position number of the scalar value is represented in the `str` member of the `value` member using a pointer to a string of decimal digit characters interpreted as a decimal number.
`vhpiHexStrVal`	The value structure represents a scalar value. The position number of the scalar value is represented in the `str` member of the `value` member using a pointer to a string of hexadecimal digit characters interpreted as a hexadecimal number.
`vhpiEnumVal`	The value structure represents an enumeration value. The enumeration value is represented in the `enumv` member of the `value` member using a value of type `vhpiEnumT`.

vhpiSmallEnumVal	The value structure represents an enumeration value. The enumeration value is represented in the smallenumv member of the value member using a value of type vhpiSmallEnumT.
vhpiIntVal	The value structure represents an integer value. The integer value is represented in the intg member of the value member using a value of type vhpiIntT.
vhpiLongIntVal	The value structure represents an integer value. The integer value is represented in the longintg member of the value member using a value of type vhpiLongIntT.
vhpiLogicVal	The value structure represents a logic value of type STD_ULOGIC or STD_LOGIC defined in the package IEEE.STD_LOGIC_1164. The logic value is represented in the enumv member of the value member using a value of type vhpiEnumT.
vhpiRealVal	The value structure represents a floating-point value. The floating-point value is represented in the real member of the value member using a value of type vhpiRealT.
vhpiStrVal	The value structure represents a string of characters. The string is represented in the str member of the value member using a pointer to a null-terminated array of characters.
vhpiCharVal	The value structure represents a character value. The character value is represented in the ch member of the value member using a value of type vhpiCharT.
vhpiTimeVal	The value structure represents a time value. The time value is represented in the time member of the value member using a time structure.
vhpiPhysVal	The value structure represents a physical value. The physical value is represented in the phys member of the value member using a physical structure.
vhpiSmallPhysVal	The value structure represents a physical value. The physical value is represented in the smallphys member of the value member using a value of type vhpiSmallPhysT.
vhpiObjTypeVal	This format is used by a VHPI program to specify that the tool provide the value of an object in a format that is appropriate for the type of the object (see 22.4).
vhpiPtrVal	The value structure represents an access value. The access value is represented in the ptr member of the value member using a pointer.
vhpiEnumVecVal	The value structure represents a one-dimensional array of enumeration values. The array value is represented in the enumvs member of the value member using a pointer to an array of values of type vhpiEnumT.
vhpiSmallEnumVecVal	The value structure represents a one-dimensional array of enumeration values. The array value is represented in the smallenumvs member of the value member using a pointer to an array of values of type vhpiSmallEnumT.
vhpiIntVecVal	The value structure represents a one-dimensional array of integer values. The array value is represented in the intgs member of the value member using a pointer to an array of values of type vhpiIntT.
vhpiLongIntVecVal	The value structure represents a one-dimensional array of integer values. The array value is represented in the longintgs member of the value member using a pointer to an array of values of type vhpiLongIntT.
vhpiLogicVecVal	The value structure represents a one-dimensional array of logic values of type STD_ULOGIC or STD_LOGIC defined in the package IEEE.STD_LOGIC_1164. The array value is represented in the enumvs member of the value member using a pointer to an array of values of type vhpiEnumT.
vhpiRealVecVal	The value structure represents a one-dimensional array of floating-point values. The array value is represented in the reals member of the value member using a pointer to an array of values of type vhpiRealT.

`vhpiTimeVecVal`	The value structure represents a one-dimensional array of time values. The array value is represented in the `times` member of the `value` member using a pointer to an array of time structures.
`vhpiPhysVecVal`	The value structure represents a one-dimensional array of physical values. The array value is represented in the `physs` member of the `value` member using a pointer to an array of physical structures.
`vhpiSmallPhysVecVal`	The value structure represents a one-dimensional array of physical values. The array value is represented in the `smallphyss` member of the `value` member using a pointer to an array of values of type `vhpiSmallPhysT`.
`vhpiPtrVecVal`	The value structure represents a one-dimensional array of access values. The array value is represented in the `ptrs` member of the `value` member using a pointer to an array of pointers.
`vhpiRawDataVal`	The value structure represents a value in the `ptr` member of the `value` member using a pointer to an implementation-defined internal representation.

An implementation may specify further formats and the way in which values are represented for those formats.

If a value structure is used by a VHPI program as an argument to the `vhpi_get_value` function and the format of the value structure specifies an array, string, or internal representation, the VHPI program shall set the `bufSize` member of the value structure to the number of bytes of storage allocated by the VHPI program for storage of the value (see 23.19).

If the format of a value structure used to represent a value specifies an array or string representation, the `numElems` member of the value structure specifies the number of elements in the array or string representation of the value represented by the value structure. If the value is represented as a string, the number of elements excludes the null termination character of the string.

If the format of a value structure used to represent a value specifies a physical type or time type representation, the `unit` member of the value structure specifies a unit of the physical or time type. The position number of the value represented by the value structure is the product of the position number of the unit and the position number of the physical or time structure or value of type `vhpiSmallPhysT` used to represent the value.

NOTE 1—A VHPI program that allocates buffer storage for a string to be written by a call to the `vhpi_get_value` function shall allow storage for the null termination character. The value written to the `bufSize` member of the value structure should be at least one more than the length of the string.

NOTE 2—The `vhpiRawDataVal` format allows a VHPI program to read the value of an object without requiring the tool to reformat the value. An implementation may allow a VHPI program to read the value of an object in its internal representation and subsequently to set the value of an object of the same type using the value, thus avoiding the performance impact of reformatting.

22.3 Reading object values

A VHPI program may read the value of certain objects in the design hierarchy information model using the `vhpi_get_value` function (see 23.19). The objects for which it is legal to read the value are:

- An object of class `name`
- An object of class `driver`
- An object of class `transaction`
- An object of class `port`
- An object of class `literal`

— An object of class `expr` for which the `Staticness` property has the value `vhpiLocallyStatic` or `vhpiGloballyStatic`

It is an error if a VHPI program uses the `vhpi_get_value` function to read the value of an object whose type is other than a scalar type or a one-dimensional array type whose element type is a scalar type, unless the format specified in the value structure is `vhpiRawDataVal` or an implementation-defined format (see 22.2.8). Furthermore, it is an error if a VHPI program uses the `vhpi_get_value` function to read the value of an object of class `name` that does not represent a locally static name.

The effect of reading the value of a given object of class `aliasDecl` is the same as the effect of reading the value of the target object of the `aliasedName` association with the given object as the reference object.

A VHPI program may read the value of an object during the elaboration phase provided the object has been elaborated. A VHPI program may read the value of a formal parameter of a subprogram provided the formal parameter has been dynamically elaborated as part of a call to the subprogram. A VHPI program may read the value of an object during the initialization and simulation phases.

For an object of class `constant`, `variable`, or `driver`, or for an object of class `signal` other than an object of class `outPort`, an object of class `portDecl` representing a port of mode **out** or an object of class `sigParamDecl` representing a signal parameter of mode **out**, the `vhpi_get_value` function yields the current value of the VHDL object represented by the object. For an object of class `outPort` or an object of class `portDecl` representing a port of mode **`out`**, the `vhpi_get_value` function yields the driving value of the VHDL object represented by the object. For an object of class `sigParamDecl` representing a signal parameter of mode **out**, the `vhpi_get_value` function yields the driving value of the driver for the signal parameter. For an object of class `transaction`, the `vhpi_get_value` function yields the value component of the transaction represented by the object.

For an object of class `file`, if the file is open, the `vhpi_get_value` function yields a string whose value is the file logical name. Otherwise, the `vhpi_get_value` function raises an error with severity `vhpiWarning`.

For an object of class `literal`, the `vhpi_get_value` function returns the value of the literal represented by the object.

For an object of class `expr`, the `vhpi_get_value` function returns the value of the expression represented by the object.

NOTE 1—A VHPI program can read the value of an object of composite type by navigating associations in the information model to acquire handles to subelements for which reading the value using the `vhpi_get_value` function is legal.

NOTE 2—A VHPI program can, as an alternative to using the `vhpi_get_value` function, read the value of an object representing a literal by reading the `IntVal`, `RealVal`, `PhysVal`, or `StrVal` property, as appropriate, of the object.

22.4 Formatting values

For each type of object whose value can be read using the `vhpi_get_value` function, there is a *native format*, defined as follows.

Object type	Native format
Any integer type	vhpiIntVal or vhpiLongIntVal
Any enumeration type other than CHARACTER, or the type STD_LOGIC or STD_ULOGIC defined in IEEE.STD_LOGIC_1164	vhpiEnumVal or vhpiSmallEnumVal
CHARACTER	vhpiCharVal
STD_LOGIC or STD_ULOGIC defined in IEEE.STD_LOGIC_1164	vhpiLogicVal
Any physical type other than TIME	vhpiPhysVal or vhpiSmallPhysVal
TIME	vhpiTimeVal
Any floating-point type	vhpiRealVal
Any access type	vhpiPtrVal
Any one-dimensional array type whose element type is an integer type	vhpiIntVecVal or vhpiLongIntVecVal
Any one-dimensional array type whose element type is an enumeration type other than CHARACTER or the type STD_LOGIC or STD_ULOGIC defined in IEEE.STD_LOGIC_1164	vhpiEnumVecVal or vhpiSmallEnumVecVal
Any one-dimensional array type whose element type is CHARACTER	vhpiStrVal
Any one-dimensional array type whose element type is STD_LOGIC or STD_ULOGIC defined in IEEE.STD_LOGIC_1164	vhpiLogicVecVal
Any one-dimensional array type whose element type is any physical type other than TIME	vhpiPhysVecVal or vhpiSmallPhysVecVal
Any one-dimensional array type whose element type is TIME	vhpiTimeVecVal
Any one-dimensional array type whose element type is any floating-point type	vhpiRealVecVal
Any one-dimensional array type whose element type is any access type	vhpiPtrVecVal

If a VHPI program calls the vhpi_get_value function with the format member of the value structure set to vhpiObjTypeVal, the function yields the value of the object formatted using the native format and updates the format member with the value of type vhpiFormatT corresponding to the native format used. For types for which there is more than one native format, the function may return the value in either format, provided the range of position numbers in the type or element type (as appropriate) is representable in the format.

A tool shall support reading of the value of an object using the native format of the object, the vhpiObjTypeVal format, and the vhpiRawDataVal format. An implementation may also support reading of the value of an object using other formats.

22.5 Updating object values

22.5.1 General

A VHPI program may update the value of certain objects in the design hierarchy information model using the `vhpi_put_value` function (see 23.28). The objects for which it is legal to update the value are:

- An object of one of the following subclasses of `objDecl`: `genericDecl`, `sigDecl`, `varDecl`, `portDecl`, `sigParamDecl`, or `varParamDecl`
- An object of class `aliasDecl` whose target object of the `aliasedName` association is an object for which it is legal to update the value
- An object of one of the following subclasses of `prefixedName`: `indexedName`, `sliceName`, or `selectedName`, provided the target object of the `prefix` association is an object for which it is legal to update the value
- An object of class `derefObj`
- An object of class `driver`
- An object of class `port`
- An object of class `funcCall`

The effect of a call to the `vhpi_put_value` function to update an object of class `genericDecl` is not specified by this standard.

The effect of updating the value of a given object of class `aliasDecl` is the same as the effect of updating the value of the target object of the `aliasedName` association with the given object as the reference object.

A VHPI program may use the `vhpi_put_value` function to update the value of the following objects during the elaboration phase provided the object to be updated has been elaborated or created:

- A signal or port of a foreign architecture
- A variable that is elaborated as part of elaboration of a shared variable, of a protected type, of a foreign architecture
- A driver created using the `vhpi_create` function
- The return value of a foreign function

A VHPI program may update the value of an object during the initialization and simulation phases. A VHPI program may update the value of a formal parameter of a subprogram provided the formal parameter is of mode **out** or **inout** and has been dynamically elaborated as part of a call to the subprogram. It is an error if a VHPI program updates the value of a formal parameter of mode **in**.

The VHPI header file defines the enumeration type `vhpiPutValueModeT` with enumeration constants corresponding to *update modes* as follows:

`vhpiDeposit`	The value of an object is updated, with no propagation of signal values.
`vhpiDepositPropagate`	The value of an object is updated, and, if the object is a signal on a net, the updated value is propagated to other signals on the net.
`vhpiForce`	An object is forced to a given value, with no propagation of signal values.
`vhpiForcePropagate`	An object is forced to a given value, and, if the object is a signal on a net, the updated value is propagated to other signals on the net.
`vhpiRelease`	The forcing of an object is released.
`vhpiSizeConstraint`	The constraint of the type of an object is set.

For objects of class other than `signal`, the effect of an update with update mode `vhpiDepositPropagate` is the same as an update with update model `vhpiDeposit`, and the effect of an update with update mode `vhpiForcePropagate` is the same as an update with update model `vhpiForce`.

If the `vhpi_put_value` function is called with an update mode of `vhpiRelease`, no value structure is required, and the value of the `value_p` argument is ignored.

It is an error if a VHPI program uses the `vhpi_put_value` function to update the value of an object whose type is other than a scalar type or a one-dimensional array type whose element type is a scalar type, unless the format specified in the value structure is `vhpiRawDataVal` or the update mode is `vhpiRelease`. Furthermore, it is an error if a VHPI program uses the `vhpi_put_value` function to update the value of an object of class `name` that does not represent a locally static name.

22.5.2 Updating an object of class variable

A call to the `vhpi_put_value` function to update the value of an object of class `variable` shall use an update mode of `vhpiDeposit`, `vhpiDepositPropage`, `vhpiForce`, or `vhpiForcePropagate`.

A call to the `vhpi_put_value` function to update the value of an object of class `variable` with an update mode of `vhpiForce` or `vhpiForcePropagate` causes the variable represented by the object to become *forced* and to be updated with the value represented by the value structure provided to the `vhpi_put_value` function. The value of a variable that is forced is not updated by a variable assignment statement or by association as an actual parameter with a formal variable parameter. The variable remains forced until a subsequent update with an update mode of `vhpiRelease`, which causes the variable to be *released*, that is, no longer to be forced.

Subelements of a variable of composite type may be separately forced. If a variable of composite type is forced, all of its subelements are forced. If a variable of composite type is released, all of the subelements of the variable are released.

For a formal variable parameter, if the parameter is passed by reference, forcing or releasing the formal parameter causes the actual parameter to be forced or released, respectively, and forcing or releasing the actual parameter causes the formal parameter to be forced or released, respectively. Otherwise, if the parameter is passed by copy, forcing or releasing the formal parameter has no effect on whether the actual parameter is forced or released, and forcing or releasing the actual parameter has no effect on whether the formal parameter is forced or released.

A call to the `vhpi_put_value` function to update the value of an object of class `variable` with an update mode of `vhpiDeposit` or `vhpiDepositPropagate` causes the variable represented by the object to be updated with the value represented by the value structure provided to the `vhpi_put_value` function, provided the variable is not forced.

NOTE—If a forced variable is updated with an update mode of `vhpiDeposit` or `vhpiDepositPropagate`, the update has no effect.

22.5.3 Updating an object of class signal

A call to the `vhpi_put_value` function to update the value of an object of class `signal` shall use an update mode of `vhpiDeposit`, `vhpiDepositPropage`, `vhpiForce`, `vhpiForcePropagate`, or `vhpiRelease`.

A call to the `vhpi_put_value` function to update the value of one of the following objects:

— An object of class `portDecl` representing a port of mode **out**

— An object of class `sigParamDecl` representing a signal parameter of mode **out**

— An object of class `outPort`

causes the driving value of the signal represented by the object to be updated; a call to update an object of class `signal` other than one of the object described in the preceding list causes the effective value of the signal represented by the object to be updated.

A call to the `vhpi_put_value` function to update the driving value of a signal with an update mode of `vhpiForce` causes the signal to become *driving-value forced*. The variable containing the driving value of the signal is updated with the value represented by the value structure provided to the `vhpi_put_value` function. Similarly, a call to the `vhpi_put_value` function to update the effective value of a signal with an update mode of `vhpiForce` causes the signal to become *effective-value forced*. The variable containing the current value of the signal is updated with the value represented by the value structure provided to the `vhpi_put_value` function.

A call to the `vhpi_put_value` function to update the driving value of a signal with an update mode of `vhpiForcePropagate` *schedules a driving-value force* for the signal, with the *driving force value* for the signal being the value represented by the value structure provided to the `vhpi_put_value` function. The effect is to cause the signal to become driving-value forced during the next signal update phase of a simulation cycle (see 14.7.3). Similarly, a call to the `vhpi_put_value` function to update the effective value of a signal with an update mode of `vhpiForcePropagate` *schedules an effective-value force* for the signal, with the *effective force value* for the signal being the value represented by the value structure provided to the `vhpi_put_value` function. The effect is to cause the signal to become effective-value forced during the next signal update phase of a simulation cycle.

If more than one driving-value force or more than one effective-value force is scheduled for a given signal before that signal is updated, the effect is not specified by this standard.

A signal that is driving-value forced remains so until a subsequent update of the signal with an update mode of `vhpiRelease`, which causes the signal to be *driving-value released*, that is, no longer to be driving-value forced, or until the signal becomes driving-value released during the signal update phase of a simulation cycle. Similarly, a signal that is effective-value forced remains so until a subsequent update of the signal with an update mode of `vhpiRelease`, which causes the signal to be *effective-value released*, that is, no longer to be effective-value forced, or until the signal becomes effective-value released during the signal update phase of a simulation cycle.

Subelements of a signal of composite type may be separately forced. If a signal of composite type is forced, all of its subelements are forced. If a signal of composite type is released, all of the subelements of the signal are released.

A call to the `vhpi_put_value` function to update the driving value of a signal with an update mode of `vhpiDeposit` causes the variable containing the driving value of the signal to be updated with the value represented by the value structure provided to the `vhpi_put_value` function, provided the signal is not driving-value forced. Similarly, a call to the `vhpi_put_value` function to update the effective value of a signal with an update mode of `vhpiDeposit` causes the variable containing the current value of the signal to be updated with the value represented by the value structure provided to the `vhpi_put_value` function, provided the signal is not effective-value forced.

A call to the `vhpi_put_value` function to update the driving value of a signal with an update mode of `vhpiDepositPropagate` *schedules a driving-value deposit* for the signal, with the *driving deposit value* for the signal being the value represented by the value structure provided to the `vhpi_put_value` function. The effect is to update the variable containing the driving value of the signal during the next signal update phase of a simulation cycle (see 14.7.3). Similarly, a call to the `vhpi_put_value` function to

update the effective value of a signal with an update mode of `vhpiDepositPropagate` *schedules an effective-value deposit* for the signal, with the *effective deposit value* for the signal being the value represented by the value structure provided to the `vhpi_put_value` function. The effect is to update the variable containing the current value of the signal during the next signal update phase of a simulation cycle.

If more than one driving-value deposit or more than one effective-value deposit is scheduled for a given signal before that signal is updated, the effect is not specified by this standard.

NOTE—If both a deposit and a force are scheduled for a given signal, the force takes precedence over the deposit. Furthermore, if a forced signal is updated with an update mode of `vhpiDeposit`, the update has no effect.

22.5.4 Updating an object of class driver

A call to the `vhpi_put_value` function to update the value of an object of class `driver` shall use an update mode of `vhpiDeposit`, `vhpiDepositPropage`, `vhpiForce`, `vhpiForcePropagate`, or `vhpiRelease`.

A call to the `vhpi_put_value` function to update the value of an object of class `driver` with an update mode of `vhpiForce` causes the driver represented by the object to become *forced*. The variable containing the current value of the driver is updated with the value represented by the value structure provided to the `vhpi_put_value` function.

A call to the `vhpi_put_value` function to update the value of an object of class `driver` with an update mode of `vhpiForcePropagate` *schedules a force* for the driver represented by the object, with the *force value* for the driver being the value represented by the value structure provided to the `vhpi_put_value` function. The effect is to cause the driver to become forced during the next signal update phase of a simulation cycle (see 14.7.3).

If more than one force is scheduled for a given driver before that driver is updated, the effect is not specified by this standard.

A driver that is forced remains so until a subsequent update of the driver with an update mode of `vhpiRelease`, which causes the driver to be *released*, that is, no longer to be forced.

A call to the `vhpi_put_value` function to update the value of an object of class `driver` with an update mode of `vhpiDeposit` causes the variable containing the current value of the driver represented by the object to be updated with the value represented by the value structure provided to the `vhpi_put_value` function, provided the driver is not forced.

A call to the `vhpi_put_value` function to update the value of an object of class `driver` with an update mode of `vhpiDepositPropagate` *schedules a deposit* for the driver represented by the object, with the *deposit value* for the driver being the value represented by the value structure provided to the `vhpi_put_value` function. The effect is to update the variable containing the current value of the driver during the next signal update phase of a simulation cycle (see 14.7.3).

If more than one deposit is scheduled for a given driver before that driver is updated, the effect is not specified by this standard.

NOTE—If both a deposit and a force are scheduled for a given driver, the force takes precedence over the deposit. Furthermore, if a forced driver is updated with an update mode of `vhpiDeposit`, the update has no effect.

22.5.5 Updating an object of class funcCall

For an object of class `funcCall` representing a function call to a foreign function, the execution function of the foreign function shall define the result returned by the function call.

If the result subtype of the function is an unconstrained type, the execution function shall set the constraint of the object of class `funcCall` using the `vhpi_put_value` function with an update mode of `vhpiSizeConstraint`, and subsequently use a call or calls to the `vhpi_put_value` function to define the result. For the call to the `vhpi_put_value` function that sets the constraint, the `numElems` member of the value structure is the number of elements in the result array. Other members of the value structure are ignored.

If the result subtype of the function is a type for which values can be represented in a single value structure, the execution function may define the result using a single call to the `vhpi_put_value` function to update the object of class `funcCall`. If the result subtype of the function is a one-dimensional array type whose element type is a scalar type, the execution function may define the result using a single call to the `vhpi_put_value` function to update the object of class `funcCall`, or may define the result using multiple calls to the `vhpi_put_value` function, as described in the following paragraph.

If the result subtype of the function is a type for which values cannot be represented in a single value structure, the execution function shall define the result using multiple calls to the `vhpi_put_value` function. The execution function shall navigate associations from the object of class `funcCall` to objects of class `name` that represent elements of the result for which values can be represented in a single value structure, and call the `vhpi_put_value` function for each such object to update the value of the element represented by the object.

A call to the `vhpi_put_value` function to define the result shall use an update mode of `vhpiDeposit`, `vhpiDepositPropage`, `vhpiForce`, or `vhpiForcePropagate`. The effect, in each case, is to update the object immediately. A call to the `vhpi_put_value` function with update mode `vhpiRelease` to define the result has no effect.

If the result subtype of the function is a composite type, it is an error if the call or calls to the `vhpi_put_value` function that define the result before the execution function returns do not define the values of all elements of the result.

An implementation may allow a VHPI program to update the value of an object of class `funcCall` representing a function call to a function other than a foreign function; the effect is not specified by this standard.

22.6 Scheduling transactions on drivers

A VHPI program may schedule a transaction on a driver or transactions on drivers in a collection of drivers using the `vhpi_schedule_transaction` function (see 23.34). The effect of *scheduling a transaction* on a driver is to modify the projected output waveform of the driver according to the rules described in 10.5.2.2. The value provided for each driver in a value structure to the `vhpi_schedule_transaction` function is used as the value component of a transaction assigned to the driver. The time component of the transaction and the delay mechanism are determined as described in 23.34.

If the `value_p` argument provided to the `vhpi_schedule_transaction` function is `NULL`, a null transaction is scheduled for the driver, or for each driver in the collection, as appropriate, represented by the object referred to by the handle provided in the `drivHdl` argument. It is an error if a null transaction is scheduled for a driver that is not a driver for a guarded signal. The effect of scheduling a null transaction on a driver defined by a sequential assignment statement or using the function `vhpi_create` is described in 10.5.2.2. The effect of using the `vhpi_schedule_transaction` function to schedule a null transaction on a driver defined by a concurrent signal assignment statement is not specified by this standard.

If the `value_p` argument is not `NULL`, it shall point to a value structure or an array of value structures that are used to specify values of transactions. The number of value structures is specified by the `numValues`

argument. In certain cases, a single value structure shall be provided, with the `numValues` argument being 1, as follows:

— If the `drivHdl` argument is a handle that refers to an object of class `driver` representing a driver for a scalar signal, the value structure shall represent a scalar value that can legally be assigned to the driver, and that value is used as the value of the transaction for the driver.

— If the `drivHdl` argument is a handle that refers to an object of class `driver` representing a driver for a resolved signal of an array type whose element type is a scalar type, the value structure shall represent an array of scalar values that can legally be assigned to the signal, and that value is used as the value of the transaction for the driver.

— If the `drivHdl` argument is a handle that refers to an object of class `driverCollection` representing a collection of drivers for elements of a signal of an array type whose element type is a scalar type, the value structure shall represent an array of scalar values, each of which can legally be assigned to an element of the signal. There shall be as many elements in the array as there are members of the collection. The value of an element of the array with a given index is used as the value of the transaction for the driver in the collection with the given index.

In other cases, an array of value structures shall be provided and is used as follows.

For a given driver, either represented by an object of class driver referred to by the handle provided as the `drivHdl` argument or in a collection of drivers represented by an object of class `driverCollection` referred to by that handle, the type of the signal driven by the driver is referred to as the *driver type*. For certain subelements of the driver type, and for the driver type itself, the value or values represented by a subarray of one or more contiguous value structures or by the entire array of value structures are formed into a *transaction subvalue* of the type of the subelement or of the driver type, respectively. The transaction subvalue for the driver type is used as the value of the transaction for the given driver.

For a subelement that is a scalar record element, the transaction subvalue is formed from the value represented by a single value structure. That value shall be a scalar value that can legally be assigned to a signal of the type of the scalar record element.

For a subelement that is an array whose element type is a scalar type, the transaction subvalue is formed from the value represented by a single value structure. That value shall be an array of scalar values, each of which can legally be assigned to a signal of the element type of the subelement. There shall be as many elements in the array as there are elements in the subelement.

For a subelement or a driver type that is an array whose element type is other than a scalar type, the transaction subvalue is formed from the concatenation of distinct subarrays corresponding to each element of the array. The subarrays occur contiguously in the array of value structures in the same order as elements in the array and are concatenated in that order to form the transaction subvalue for the array.

For a subelement or driver type that is a record, the transaction subvalue is formed from the concatenation of distinct subarrays corresponding to each element of the record. The subarrays occur contiguously in the array of value structures in the same order as the order in which the elements are declared in the record type definition for the type of the subelement or driver type, as appropriate, and are concatenated in that order to form the transaction subvalue for the array.

If the `drivHdl` argument is a handle that refers to an object of class `driver`, the array of value structures is used to form the transaction subvalue for the driver type of the driver represented by the object, and the transaction subvalue is used as the value of the transaction for the driver. It is an error if the number of value structures is insufficient to form the transaction subvalue.

If the `drivHdl` argument is a handle that refers to an object of class `driverCollection`, a transaction subvalue is formed from a distinct subarray for each member of the collection represented by the object. The

subarrays occur contiguously in the array of value structures in the same order as the order in which the members occur in the collection. The transaction subvalue for each member is used as the value of the transaction for the member. It is an error if the number of value structures is insufficient to form the transaction subvalues.

NOTE—An object of class `driver` represents a driver for a basic signal.

23. VHPI function reference

23.1 General

This clause describes each of the functions in the VHPI. It describes the arguments required to be passed to each function, the operation performed by the function, and the result value returned by the function to a VHPI program.

Where a given VHPI function called by a thread of control in a VHPI program returns a pointer to a string or a structure, either as the result of the function or in a location pointed to by an argument of the function, the string or structure is either *permanent* or *transient*. Unless otherwise specified, the default is for such a string or structure to be transient. In the case of a string, the string is represented as a null-terminated array of characters. A permanent string or structure is allocated by the tool in storage that is not subsequently overwritten during the invocation of the tool. A VHPI program may store a pointer to a permanent string or structure for subsequent reference to the string or structure. A transient string or structure is allocated by the tool in storage that may subsequently be overwritten. The value of the string or structure persists at least until the earlier of

— the next call to the given VHPI function by the same thread of control, or
— return to the tool by the thread of control that called the given VHPI function.

If a VHPI program needs to refer to the value of a transient string or structure beyond the interval for which it persists, the VHPI program shall copy the value.

23.2 vhpi_assert

Reports an error message.

Synopsis:

```
int vhpi_assert(vhpiSeverityT severity, char *formatmsg, ...);
```

Description:

The `vhpi_assert` function performs an operation that is equivalent to the VHDL report statement. The character string pointed to by the `formatmsg` argument is a format string that may contain conversion codes as defined for the C `printf` function in ISO/IEC 9899:1999/Cor 1:2001. The format string and subsequent arguments to the `vhpi_assert` function are interpreted in the same way as specified in ISO/IEC 9899:1999/Cor 1:2001 for the C `printf` function to form a formatted character string that corresponds to the string expression value in a report statement, and the value of the `severity` argument corresponds to the severity expression value in report statement.

Return value:

0 if the operation completes without error, or 1 otherwise.

NOTE—Execution of the `vhpi_assert` function may cause a simulation to stop, depending on the value of the `severity` argument and on the simulator.

Example:

In the following VHPI program, the `vhpi_assert` function is used to report an error message if the value of a signal named `clk` is not '1'.

```
int check_clock_signal(vhpiHandleT scopeHdl) {
  vhpiHandleT clkHdl;
  vhpiValueT value;
  /* look up a VHDL object of name clk at the scope instance */
  /* get a handle to the clk named object */
  clkHdl = vhpi_handle_by_name("clk", scopeHdl);
  if (!clkHdl) return 1;
  value.format = vhpiLogicVal;
  vhpi_get_value(clkHdl, &value);
  if (value.logic == vhpiBit0) {
    vhpi_assert(vhpiError, "clock not high: %d", value.logic);
    return 1;
  }
  return 0;
}
```

23.3 vhpi_check_error

Retrieves information about an error raised by a VHPI function.

Synopsis:

```
int vhpi_check_error (vhpiErrorInfoT *error_info_p);
```

Description:

The `vhpi_check_error` function checks whether the immediately previous call to a VHPI function raised an error. The `error_info_p` argument is either a pointer to an *error information structure* in which error information is returned or `NULL`. If the value of `error_info_p` is not `NULL`, the `vhpi_check_error` function writes information about the error into the error information structure. Memory for the structure shall be allocated by the VHPI program that calls `vhpi_check_error` before the call.

If no error was raised by the previous call to a VHPI function and the value of the `error_info_p` is not `NULL`, the values written into members of the structure, if any, are not specified. Otherwise, if an error occurred and the value of the `error_info_p` is not `NULL`, the members of the structure are written as follows:

- `severity`: The severity level of the error.
- `message`: A pointer to a string that describes the error.
- `str`: A pointer to a string whose content is implementation defined.
- `file`: A pointer to a string containing the name of the VHDL source file that contains the VHDL item corresponding to the VHPI handle passed to the VHPI function that raised the error; or `NULL` if no such VHDL source file can be identified.
- `line`: The number of the line in the VHDL source file containing the VHDL item corresponding to the VHPI handle passed to the VHPI function that raised the error; or –1 if no such line can be identified.

Return value:

0 if no error occurred on the previous call to a VHPI function, or 1 otherwise.

NOTE 1—An implementation might use the `str` member of the error information structure to return such information as a mnemonic abbreviation of the error description or the name of a product that raised the error.

NOTE 2—An implementation may provide error information in a log file or a standard output stream. Such provision is independent of the use of the `vhpi_check_error` by a VHPI program function to retrieve error information.

Examples:

In the following VHPI program, the `vhpi_check_error` function is used to determine whether a previous function raised an error. If it did, the severity information provided by the `vhpi_check_error` function is used to determine what recovery action to take.

```
vhpiErrorInfoT err;

if (vhpi_check_error(&err)) {
  switch (err.severity) {
  case vhpiError:
  case vhpiFailure:
  case vhpiInternal:
    return;
  case vhpiSystem:
    if (errno == ...)
      return;
    break;
  default:
    /* examine and decide if need termination */
    ...
  }
}
```

Given the following VHDL model in the file `myvhdl.vhd`

```
entity TOP is
end TOP;

architecture MY_VHDL of TOP is
   constant VAL: INTEGER := 0;
   signal S1, S2, S3: BIT;
begin
   u1: C_AND (S1, S2, S3);
   process (S1)
      variable VA: INTEGER:= VAL;
   begin
      VA := MYFUNC(S1);
   end process;
end MY_VHDL;
```

the following VHPI program uses the `vhpi_check_error` function to determine whether the call to `vhpi_iterator` succeeded. The VHPI program also uses the `vhpi_check_error` function to check whether the call to `vhpi_handle` succeded during each iteration of the while loop. If the call raised an error with severity greater than `vhpiWarning`, the VHPI program uses the file name and line number information, if provided, in an error message.

```
/* hdl is a handle to the root instance */
void traverse_hierarchy(vhpiHandleT hdl) {
  vhpiHandleT subHdl, itr, duHdl;
  vhpiErrorInfoT err;
```

```
  itr = vhpi_iterator(vhpiInternalRegions, hdl);
  /* if error code is != 0 do not continue */
  if (vhpi_check_error(NULL)) return;

  if (itr)
    while (subHdl = vhpi_scan(itr)) {
      duHdl = vhpi_handle(vhpiDesignUnit, subHdl);
      if (vhpi_check_error(&err)) {
        if (err.severity > vhpiWarning)
          if (err.file != NULL)
            vhpi_printf("An error occurred during call to "
                        "traverse_hierarchy at filename %s line %d\n",
                        err.file, err.line);
          else
            vhpi_printf("An error occurred during call to "
                        "traverse_hierarchy\n");
        return;
      }
      switch (vhpi_get(vhpiKindP, subHdl)) {
        ...
      }
    }
}
```

Since the internal region of the process object in the information model does not have a one-to-one association with a design unit object, the VHPI program produces the following output:

```
An error occurred during call to traverse_hierarchy at file myvhdl.vhd line 8
```

23.4 vhpi_compare_handles

Compares handles.

Synopsis:

```
int vhpi_compare_handles (vhpiHandleT handle1, vhpiHandleT handle2);
```

Description:

Determines whether the arguments `handle1` and `handle2` refer to the same object.

Return value:

1 if handle1 and handle2 refer to the same object, or 0 otherwise.

NOTE—Handle equivalence cannot be checked with the C comparison operator ==, since two handles with different representations may nonetheless refer to the same object.

Example:

The following function in a VHPI program searches for a declaration of a signal named `clk` in a given scope. It uses the `vhpi_compare_handles` function to compare a handle to an object named `clk` with handles to successive signal declarations in the scope.


```
vhpiHandleT find_clock_signal(vhpiHandleT scopeHdl) {
  vhpiHandleT sigHdl, clkHdl, itrHdl;
  int found = 0;

  clkHdl = vhpi_handle_by_name("clk", scopeHdl);
  itrHdl = vhpi_iterate(vhpiSigDecl, scopeHdl);
  while (sigHdl = vhpi_scan(itrHdl)) {
    if (vhpi_compare_handles(sigHdl, clkHdl)) {
      found = 1;
      break;
    } else
      vhpi_release_handle(sigHdl);
  }
  vhpi_release_handle(itrHdl);
  if found
    return(sigHdl);
  else
    return(NULL);
}
```

23.5 vhpi_control

Issues a control request to the VHPI tool.

Synopsis:

```
int vhpi_control (vhpiSimControlT command, ...);
```

Description:

The value of the `command` argument specifies the control action requested. Subsequent arguments specify additional information required by the tool to perform the control action.

This standard specifies three control actions, corresponding to the enumeration values `vhpiStop`, `vhpiFinish`, and `vhpiReset` of type `vhpiSimControlT`. If a tool implements any of these control actions, the effect shall be as follows:

- If `command` is `vhpiStop`, after control returns to the tool from the callback function from which the `vhpi_control` function was invoked, the tool stops simulation then accepts further directives from an interactive user or a command source. Additional arguments to `vhpi_control` beyond the `command` argument may be interpreted by the tool in an implementation-defined manner. The tool shall provide an implementation-defined default action if no additional arguments are provided.
- If `command` is `vhpiFinish`, after control returns to the tool from the callback function from which the `vhpi_control` function was invoked, the tool enters the termination phase (see 20.10). Additional arguments to `vhpi_control` beyond the command argument may be interpreted by the tool in an implementation-defined manner. The tool shall provide an implementation-defined default action if no additional arguments are provided.
- If `command` is `vhpiReset`, after control returns to the tool from the callback function from which the `vhpi_control` function was invoked, the tool enters the reset phase (see 20.9). Additional arguments to `vhpi_control` beyond the command argument may be interpreted by the tool in an implementation-defined manner. The tool shall provide an implementation-defined default action if no additional arguments are provided.

For each of these control actions, if implemented, the number of steps of the simulation cycle performed by the tool between return of control to the tool and the tool performing the requested control action is implementation defined, except that no new simulation cycle is commenced before the control action is performed.

If `command` is a value other than one of `vhpiStop`, `vhpiFinish`, or `vhpiReset`, the tool performs an implementation-defined control action, which may make use of additional arguments beyond the `command` argument. The tool may perform the requested control action immediately or may queue the request to be performed at an implementation-defined time after control returns to the tool from the callback function from which the `vhpi_control` function was invoked.

If a VHPI program calls the `vhpi_control` function before the tool has performed a control action requested by a prior call to the function, the order in which the control actions are performed is implementation defined, except that control actions corresponding to `vhpiStop`, `vhpiFinish`, and `vhpiReset` are performed in the order in which they are requested.

Return value:

0 if the operation completes without error, or 1 otherwise.

Errors:

It is an error if `vhpi_control` is called with the `command` argument having the value `vhpiStop`, `vhpiFinish`, or `vhpiReset` while the tool is any execution phase other than the simulation phase.

If a tool does not implement a control action requested using `vhpi_control`, the `vhpi_control` function shall raise an error.

NOTE—In response to a call to `vhpi_control` with the argument `vhpiFinish`, the tool does not perform any `vhpiCbEndOfSimulation` callbacks.

Example:

The following VHPI program performs some operations and then calls `vhpi_control` with `vhpiFinish` to terminate tool execution.

```
void user_app() {
  /* Application traverse hierarchy */
  ...
  /* Application collect information */
  ...
  vhpi_control(vhpiFinish);
}
```

23.6 vhpi_create

Creates an object of class `processStmt`, `driver`, `driverCollection`, or `anyCollection`; or appends an object to a collection.

Synopsis:

```
vhpiHandleT vhpi_create (vhpiClassKindT kind,
                         vhpiHandleT handle1, vhpiHandleT handle2);
```


Description:

The kind argument specifies the class of object to be created.

If the value of kind is vhpiProcessStmtK, handle1 shall refer to an object of class archBody whose IsForeign property has the value vhpiTrue, and handle2 shall be NULL. The function creates an object of class processStmt associated with the object referred to by handle1.

If the value of kind is vhpiDriverK, handle1 shall refer to an object of class basicSignal, and handle2 shall either refer to an object of class processStmt whose IsForeign property has the value vhpiTrue or be NULL. The function creates an object of class driver associated with the object referred to by handle1. The value of the IsForeign property of the created object is vhpiTrue. If handle2 is not NULL, the object of class driver is also associated with the object referred to by handle2; otherwise the object of class driver is not associated with a target object of class processStmt.

For an object of class processStmt created by a call to the vhpi_create function, the value of the IsForeign property is vhpiTrue, the values of the IsPassive and IsPostponed properties are vhpiFalse. The values of properties representing line numbers are vhpiUndefined. The values of name properties are not specified by this standard. Associations representing declarations, specifications, statements, and the sensitivity list of the process have no target objects.

If the value of kind is vhpiDriverCollectionK, handle1 shall either be NULL or refer to an object of class driverCollection, and handle2 shall refer to an object of class driver or driverCollection. If handle1 is NULL, the function creates a new collection object of class driverCollection and appends one or more objects of class driver as members to the collection. If handle1 is not NULL, the function appends one or more objects of class driver as members to the collection object referred to by handle1. In either case, if handle2 refers to an object of class driver, that object is the single object appended as a member to the collection by the function. Otherwise, if handle2 refers to an object of class driverCollection, all of the members in the collection referred to by handle2 are appended, in the order in which they occur in the collection referred to by handle2, to the new collection or to the collection referred to by handle1.

If the value of kind is vhpiAnyCollectionK, handle1 shall either be NULL or refer to an object of class anyCollection, and handle2 shall refer to an object of any class. If handle1 is NULL, the function creates a new collection object of class anyCollection and appends one or more objects as members to the collection. If handle1 is not NULL, the function appends one or more objects as members to the collection object referred to by handle1. In either case, if handle2 refers to an object of some class other than collection, that object is the single object appended as a member to the collection by the function. Otherwise, if handle2 refers to an object of class collection, all of the members in the collection referred to by handle2 are appended, in the order in which they occur in the collection referred to by handle2, to the new collection or to the collection referred to by handle1.

Return value:

A handle to the newly created object or collection or to the augmented collection, as appropriate, if the operation completes without error, or NULL otherwise.

Errors:

It is an error if vhpi_create is called with kind having the value vhpiProcessStmtK or vhpiDriverK other than during the elaboration, initialization, or simulation phases of tool execution (see Clause 20).

It is an error if `vhpi_create` is called with kind having the value `vhpiDriverK` and `handle2` being not `NULL`, and the process represented by the object referred to by `handle2` already has a driver for the basic signal represented by the object referred to by `handle1`.

It is an error if the members of a collection object of class `driverCollection` are not all drivers of subelements of the same declared signal.

Example:

In the following VHPI program, the function `vhpi_create` is used to create a process in a foreign architecture and to create a driver for each signal declared in the architecture.

```
void create_vhpi_driver(vhpiHandleT archHdl) {
  vhpiHandleT drivHdl, sigItr, sigHdl, processHdl;
  vhpiHandleT arr_driv[MAX_DRIVERS];
  int i = 0;

  if (!vhpi_get(vhpiIsForeignP, archHdl))
    return;
  /* create a VHPI process */
  processHdl = vhpi_create(vhpiProcessK, archHdl, NULL);
  /* iterate on the signals declared in the architecture and create a
     VHPI driver for each of them */
  sigItr = vhpi_iterator(vhpiSigDecls, archHdl);
  if (!sigItr) return;
  while (sigHdl = vhpi_scan(sigItr)) {
    drivHdl = vhpi_create(vhpiDriverK, sigHdl, processdl);
    arr_driv[i] = drivHdl;
    i++;
  }
}
```

In the following VHPI program, the function `vhpi_create` is used to create a collection of drivers for the basic signals of a signal.

```
void create_vhpi_collection(vhpiHandleT sigHdl) {
  vhpiHandleT itBasic, basicH, itDriver, driverH;
  vhpiHandleT h = NULL;

  itBasic =  vhpi_iterator(vhpiBasicSignals, sigHdl);
  while (basicH = vhpi_scan(itBasic)) {
    itDriver = vhpi_iterator(vhpiDrivers, basicH)
    while (driverH = vhpi_scan(itDriver) {
      h = vhpi_create(vhpiDriverCollectionK, h, driverH);
    }
  }
}
```

23.7 vhpi_disable_cb

Disables a registered callback.

Synopsis:

```
int vhpi_disable_cb (vhpiHandleT cb_obj);
```

Description:

If the object referred to by the `cb_obj` argument is an enabled callback, the function disables it, thus preventing call of the callback function when the callback trigger event occurs.

Return value:

0 if the operation completes without error, or 1 otherwise.

Errors:

If the object is a disabled or mature callback, the function leaves the callback unchanged and raises an error condition with severity `vhpiWarning`.

See also:

`vhpi_register_cb`, `vhpi_enable_cb`, `vhpi_get_cb_info`, `vhpi_remove_cb`.

23.8 vhpi_enable_cb

Enables a registered callback.

Synopsis:

```
int vhpi_enable_cb (vhpiHandleT cb_obj);
```

Description:

If the object referred to by the `cb_obj` argument is a disabled callback, the function enables it, thus allowing call of the callback function when the callback trigger event occurs.

Return value:

0 if the operation completes without error, or 1 otherwise.

Errors:

If the object is an enabled or mature callback, the function leaves the callback unchanged and raises an error condition with severity `vhpiWarning`.

See also:

`vhpi_register_cb`, `vhpi_disable_cb`, `vhpi_get_cb_info`, `vhpi_remove_cb`.

Example:

In the following VHPI program, the function `vhpi_enable_cb` is used to enable a callback that was registered but is disabled.

```
static vhpiHandleT mylastcbk = 0;

void activate_cbk(vhpiHandle cbHdl) {
  vhpiStateT cbState;
  cbState = vhpi_get(vhpiStateP, cbHdl);
  if (cbState == vhpiDisable)
    vhpi_enable_cb(cbHdl);
}

void register_cbk() {
  vhpiCbDataT cbData;
  vhpiHandleT cbHdl;
  int flags;
  flags = vhpiDisableCb | vhpiReturnCb;
  cbData.reason = vhpiCbEndOfSimulation;
  cbData.cb_rtn = myf;
  cbHdl = vhpi_register_cb(&cbData, flags);
  mylastcbk = cbHdl;
}

int main (int argc, char *argv[] ){
  register_cbk();
  ...
  activate_cbk(mylastcbk);
  return(0);
}
```

23.9 vhpi_format_value

Changes the format used to represent a value.

Synopsis:

```
int vhpi_format_value (vhpiValueT *in_value_p,
                       vhpiValueT *out_value_p);
```

Description:

The `in_value_p` argument is a pointer to a value structure, referred to in this description as the *input value structure*, representing the value to be represented in a new format. The `out_value_p` argument is a pointer to a value structure, referred to in this description as the *output value structure*, specifying the new format and containing storage into which the newly formatted value is written. Storage for both value structures is allocated by the VHPI program before calling the function.

The function converts the value that is represented in the input value structure to the format specified in the `format` member of the output value structure. If the newly formatted value is a scalar, the function writes the newly formatted value to the `value` member of the output value structure. If the newly formatted value is represented as an array, string, or using internal representation and the value of the `value` member of the output value structure is `NULL`, the function does not write the newly formatted value, but returns the minimum number of bytes of storage that would be required to write the value.

If the newly formatted value is represented as an array, string, or using internal representation, the VHPI program, before calling `vhpi_format_value`, shall allocate storage for the newly formatted value and

shall write the size in bytes and the address of the storage into the `bufSize` and `value` members, respectively, of the output value structure. In that case, the function writes the newly formatted value to the storage pointed to by the `value` member of the output value structure (see 22.2).

If the newly formatted value is represented as a physical or time value or an array of physical or time values, the VHPI program, before calling `vhpi_format_value`, shall write the position number of a scale factor into the `unit` member of the output value structure.

The value format conversions that can be performed by the `vhpi_format_value` function are implementation defined.

Return value:

0 if the newly formatted value is a scalar and the operation completes without error; or the minimum size in bytes of storage required to represent the value in the specified format if the newly formatted value is represented as an array, string, or using internal representation and either the `value` member of the output value structure is `NULL` or the size provided in the `bufSize` member of the output value structure is insufficient; or a negative integer otherwise.

Errors:

It is an error if either `in_value_p` or `out_value_p` is `NULL`. It is an error if the newly formatted value is outside of the range of values that can be represented. It is an error if the combination of the `format` members of the input and output value structures specify a value format conversion that cannot be performed.

It is an error if the amount of storage allocated for a value represented as an array, string, or using internal representation is insufficient for the newly formatted value.

See also:

`vhpi_get_value`.

Example:

In the following VHPI program, the `vhpi_format_value` function is called first to convert a real value to an integer value, and second to convert a time value from a precision of fs to a precision of ns.

```
vhpiValueT value, newValue;
vhpiValueT * valuep, newValuep;
vhpiErrInfoT errInfo;

valuep = &value;
newValuep = &newValue;
value.format = vhpiRealVal;
if (vhpi_get_value(objHdl, valuep))
  vhpi_check_error(&errInfo);
newValue.format = vhpiIntVal;
if (vhpi_format_value(valuep, newValuep))
  vhpi_check_error(&errInfo);

value.format = vhpiTimeVal;
vhpi_get_value(objHdl, valuep);
newValue.unit = vhpiNS; /* physical position of ns */
```

```
newValue.format = vhpiTimeVal;
if (vhpi_format_value(valuep, newValuep))
  vhpi_check_error(&errInfo);
```

23.10 vhpi_get

Gets the value of an integer or Boolean property of an object.

Synopsis:

```
vhpiIntT vhpi_get(vhpiIntPropertyT property, vhpiHandleT object);
```

Description:

The `property` argument is an enumeration constant that corresponds to an integer or Boolean property. The `object` argument is a handle to an object that has the corresponding integer or Boolean property. The function reads the value of the property of the object.

Return value:

The value of the property if the property can be read, or `vhpiUndefined` otherwise.

See also:

`vhpi_get_phys`, `vhpi_get_real`, `vhpi_get_str`.

NOTE—Some integer properties may legally have the same value as `vhpiUndefined`. In such cases, a VHPI program should use the `vhpi_check_error` function to determine whether an error was raised by `vhpi_get` rather than simply testing the return value of `vhpi_get`.

23.11 vhpi_get_cb_info

Gets information about a registered callback.

Synopsis:

```
int vhpi_get_cb_info (vhpiHandleT object, vhpiCbDataT *cb_data_p);
```

Description:

The `object` argument is a handle to an object of class `callback`. The `cb_data_p` argument is a pointer to a callback data structure. The VHPI program calling `vhpi_get_cb_info` shall allocate memory for the callback data structure before the call.

The function retrieves information about the callback object referred to by `object` and writes the information into the callback data structure pointed to by `cb_data_p`. The information returned in the callback data structure is equivalent to that provided in a callback data structure to the `vhpi_register_cb` function when the callback was registered. The values of the `reason`, `cb_rtn`, and `user_data` members of the callback data structure written by `vhpi_get_cb_info` are the same as the values of the `reason`, `cb_rtn`, and `user_data` members, respectively, of the registration callback data structure.

If the registration callback data structure included a valid handle in the `obj` member, the `obj` member of the callback data structure written by `vhpi_get_cb_info` is a handle that refers to the same object as

that referred to by the obj member of the registration callback data structure; otherwise, the value of the obj member of the callback data structure written by vhpi_get_cb_info is not specified.

If the registration callback data structure included a pointer to a time structure in the time member, the time member of the callback data structure written by vhpi_get_cb_info is a pointer to a time structure, allocated by the tool, with the same value as the time structure pointed to by the time member of the registration callback data structure; otherwise, the time member of the callback data structure written by vhpi_get_cb_info is NULL.

If the registration callback data structure included a pointer to a value structure in the value member, the value member of the callback data structure written by vhpi_get_cb_info is a pointer to a value structure, allocated by the tool, with the same value as the value structure pointed to by the value member of the registration callback data structure; otherwise, the value member of the callback data structure written by vhpi_get_cb_info is NULL.

Return value:

0 if the operation completes without error, or 1 otherwise.

Errors:

A VHPI program that releases a handle that is the value of the obj member of the callback data structure written by vhpi_get_cb_info is erroneous.

See also:

vhpi_register_cb, vhpi_enable_cb, vhpi_disable_cb, vhpi_remove_cb.

23.12 vhpi_get_data

Gets saved data for restart.

Synopsis:

```
size_t vhpi_get_data(int32_t id,
                     void * dataLoc, size_t numBytes);
```

Description:

The id argument is an identification number for a saved data set. The dataLoc argument is the address to which data read from the saved data set is written. The numBytes argument is the number of bytes of data to read.

The function reads a number of bytes, given by numBytes, from the saved data set identified by id and writes the data to the address pointed to by dataLoc. The VHPI program calling vhpi_get_data shall allocate storage pointed to by dataLoc before the call.

The first call to vhpi_get_data with a given value for id during a given occurrence of the restart phase of tool execution reads bytes from the saved data set starting from the first location of the saved data set. Subsequent calls to vhpi_get_data with the same id value during the same occurrence of the restart phase read bytes starting from the location immediately after the last location read by the immediately preceding call with the given id value.

If a data set contains unread bytes of data, a call to `vhpi_get_data` reads the lesser of `numBytes` of data or the number of unread bytes that remain. If fewer than `numBytes` bytes remain, the bytes of storage pointed to by `dataLoc`, beyond those written with read data and up to a total of `numBytes` bytes of data in total, are written with the value 0.

A VHPI program may read fewer bytes of a saved data set than were saved in the data set.

Return value:

The number of bytes actually read, or 0 if the read failed.

Errors:

It is an error if `vhpi_get_data` is called other than from a `vhpiCbStartOfRestart` or `vhpiCbEndOfRestart` callback.

It is an error if the `id` value is not valid for the occurrence of the restart phase of tool execution during which the `vhpi_get_data` function is called.

If fewer than `numBytes` bytes remain to be read, the `vhpi_get_data` function raises an error condition with severity `vhpiWarning`.

See also:

`vhpi_put_data`.

NOTE—Since a call to `vhpi_get_data` may read fewer bytes than requested, the VHPI program should check the number of bytes actually read rather than assuming all requested bytes are read.

Example:

In the following VHPI program, the `vhpi_get_data` function is used first to read the number of linked list elements in a saved data set and second to read that number of linked list elements. The VHPI program function that calls `vhpi_get_data` is a `vhpiCbStartOfRestart` callback (see example in 23.27).

```
/* type definitions for private data structures to save used by the
   foreign models or applications */
struct myStruct{
  struct myStruct *next;
  int d1;
  int d2;
}

void consumer_restart(vhpiCbDataT *cbDatap) {
  int status;
  int cnt = 0;
  struct myStruct *wrk;
  int dataSize = 0;

  /* get the id for this restart callback */
  int id = (int) cbDatap->user_data;
  /* get the number of structures */
  status = vhpi_get_data(id, (char *)&cnt, sizeof(int));
  if (status != sizeof(int))
    vhpi_assert(vhpiError, "Data read is not an int %d\n", status);
```



```
  /* allocate memory to receive the data that is read in */
  firstWrk = calloc(cnt, sizeof(struct myStruct));

  /* retrieve the data for the first structure */
  dataSize = cnt * sizeof(struct myStruct);
  status = vhpi_get_data(id, (char *)wrk, dataSize);
  if (status != dataSize)
    vhpi_assert(vhpiError, "Cannot read %d data structures\n", cnt );

  /* fix up the next pointers in the link list:
     recreate the linked list */
  for (wrk = firstWrk; cnt >0; cnt--) {
    wrk->next = wrk++;
    wrk = wrk->next;
  }
} /* end of consumer_restart */
```

23.13 vhpi_get_foreignf_info

Gets information about a foreign model or application.

Synopsis:

```
int vhpi_get_foreignf_info (vhpiHandleT hdl,
                            vhpiForeignDataT *foreignDatap);
```

Description:

The `hdl` argument is a handle to an object of class `foreignf`, and the `foreignDatap` argument is a pointer to a foreign data structure. The function retrieves information about the foreign model or application represented by the object referred to by `hdl` and writes the information into the foreign data structure pointed to by `foreignDatap`. The VHPI program calling `vhpi_get_foreignf_info` shall allocate memory for the foreign data structure before the call.

The value of the `kind` member identifies whether the object referred to by `hdl` is a foreign architecture, function, procedure, or application (see 20.2). If the object referred to by `hdl` is a foreign architecture and an elaboration function was specified during registration of the foreign architecture, the value of the `elabf` member is a pointer to the elaboration function; otherwise the value of the `elabf` member is `NULL`. The value of the `execf` member is a pointer to the execution or registration function, as appropriate, specified during registration for the object referred to by `hdl`.

The value of the `libraryName` member is a pointer to a permanent string whose value is the object library path denoting the physical object library, identified during registration, that contains the entry points for the foreign model or application.

The value of the `modelName` member is a pointer to a permanent string whose value is the model name or application name, as appropriate, of the foreign model or application. If the object referred to by `hdl` is a foreign model that was registered other than using standard direct binding (20.2.4.3), the model name of the foreign model is the model name specified during registration of the foreign model. If the object referred to by `hdl` is a foreign model that was registered using standard direct binding, the model name of the foreign model is the simple name of the architecture body or the designator of the subprogram, as appropriate, of the foreign model. If the object referred to by `hdl` is a foreign application, the value of the `modelName`

member is a pointer to a permanent string whose value is the application name specified during registration of the foreign application.

Return value:

0 if the operation completes without error, or 1 otherwise.

Errors:

If the tool has registered but not bound the elaboration, execution, or registration function of a foreign model or application or library of foreign models when a VHPI program calls `vhpi_get_foreignf_info` with a handle referring to the foreign model or application or library of foreign models, the `vhpi_get_foreignf_info` function raises an error with severity `vhpiWarning`.

See also:

`vhpi_register_foreignf`, `vhpi_iterator(vhpiForeignfs, NULL)`.

23.14 vhpi_get_next_time

Gets the time of the next simulation cycle.

Synopsis:

```
int vhpi_get_next_time (vhpiTimeT  *time_p);
```

Description:

The `time_p` argument is a pointer to a time structure in which to write the time of the next simulation cycle. The time structure shall be allocated by the VHPI program that calls `vhpi_get_next_time` before the call. The function writes to the time structure the value of T_n, the time of the next simulation cycle (see 14.7.5.1).

Return value:

`vhpiNoActivity` if T_n = TIME'HIGH and there are no active drivers, process resumptions, or registered and enabled `vhpiCbAfterDelay`, `vhpiCbRepAfterDelay`, `vhpiCbTimeOut`, or `vhpiCbRepTimeOut` callbacks to occur at T_n; a non-zero value other than `vhpiNoActivity` if an error occurs; or 0 otherwise.

Errors:

`vhpi_get_next_time` shall be called during step m) (see 14.7.5.2) of the initialization phase or during the simulation phase of model execution. It is an error if it is called at any other time.

See also:

`vhpi_get_phys(vhpiResolutionLimitP, NULL)`, `vhpi_get_time`.

NOTE 1—A VHPI program can use the `vhpi_format_value` function to change the way in which the time value is expressed.

NOTE 2—If the next simulation cycle is a delta cycle, the time of the next simulation cycle is the same as the current simulation time.

Example:

In the following VHPI program, the function `vhpi_get_next_time` is used to get the time of the next simulation cycle for display in an informative message.

```
vhpiTimeT time;

switch (vhpi_get_next_time(&time)) {
case vhpiNoActivity:
  vhpi_printf("simulation is over, %d %d\n", time.high, time.low);
  break;
case 0:
  vhpi_printf("time = %d %d\n", time.high, time.low);
  break;
default:
  vhpi_check_error(&errInfo);
  break;
}
```

23.15 vhpi_get_phys

Gets the value of a physical property of an object.

Synopsis:

```
vhpiPhysT vhpi_get_phys (vhpiPhysPropertyT property,
                         vhpiHandleT object);
```

Description:

The `property` argument is an enumeration constant that corresponds to a physical property. The `object` argument is a handle to an object that has the corresponding physical property. The function reads the value of the property of the object.

Return value:

The value of the property if the property can be read, or an unspecified value otherwise.

See also:

`vhpi_get`, `vhpi_get_real`, `vhpi_get_str`.

Example:

In the following VHPI program, the `vhpi_get_phys` function is used to read the right bound of the range constraint of a physical type declaration.

```
vhpiHandleT type; /* a physical type declaration */;
vhpiHandleT range = vhpi_handle(vhpiConstraint, type);
vhpiPhysT phys = {0,0};

phys = vhpi_get_phys(vhpiPhysRightBoundP, range));
vhpi_printf(" right bound of physical type is %d %d \n",
            phys.low, phys.high);
```

23.16 vhpi_get_real

Gets the value of a real property of an object.

Synopsis:

```
vhpiRealT vhpi_get_real (vhpiRealPropertyT property, vhpiHandleT
object);
```

Description:

The `property` argument is an enumeration constant that corresponds to a real property. The `object` argument is a handle to an object that has the corresponding real property. The function reads the value of the property of the object.

Return value:

The value of the property if the property can be read, or an unspecified value otherwise.

See also:

`vhpi_get`, `vhpi_get_phys`, `vhpi_get_str`.

Example:

In the following VHPI program, the `vhpi_get_real` function is used to read the right bound of the range constraint of a floating-point type declaration.

```
vhpiHandleT type; /* a float type declaration */;
vhpiHandleT range = vhpi_handle(vhpiConstraint, type);

vhpi_printf(" right bound of floating type is %f\n",
            vhpi_get_real(vhpiFloatRightBoundP, range));
```

23.17 vhpi_get_str

Gets the value of a string property of an object.

Synopsis:

```
const vhpiCharT * vhpi_get_str (vhpiStrPropertyT property,
                                vhpiHandleT object);
```

Description:

The `property` argument is an enumeration constant that corresponds to a string property. The `object` argument is a handle to an object that has the corresponding string property. The function reads the value of the property of the object.

Return value:

A pointer to a string that is the value of the property, if the property can be read, or `NULL` otherwise.

See also:

`vhpi_get`, `vhpi_get_phys`, `vhpi_get_real`.

NOTE—Some string property values may include special characters (for example, the character \ in an extended identifier). VHPI programs that use such property values should ensure that special characters are not inadvertently misinterpreted, for example, as escape characters, in subsequent operations.

Example:

In the following VHPI program, the `vhpi_get_str` function is used to read the name of the definition of a component instance.

```
char name[MAX_LENGTH];
vhpiHandleT inst = vhpi_handle_by_name(":u1", NULL);

strcpy(name, vhpi_get_str(vhpiDefNameP, inst));
vhpi_printf("instance u1 is a %s\n", name);
```

23.18 vhpi_get_time

Gets the current simulation time.

Synopsis:

```
void vhpi_get_time (vhpiTimeT  *time_p, long *cycles);
```

Description:

The `time` argument is a pointer to a time structure in which to write the current simulation time or `NULL`. The cycles argument is a pointer to location in which to write the number of delta cycles or `NULL`. The VHPI program calling `vhpi_get_time` shall allocate memory for the time structure and number of delta cycles, if required, before the call.

If the `time` argument is not NULL, the function writes the current simulation time to the time structure.

If the `cycles` argument and the `time` argument are both not `NULL`, the function writes the number of delta cycles that have occurred at the current time, T_c, to the location pointed to by the `cycles` argument. If the `cycles` argument is not `NULL` and the `time` argument is `NULL`, the function writes the total number of simulation cycles that have occurred in the current invocation of the simulation phase of tool execution to the location pointed to by the `cycles` argument. In either case, the number is expressed as a value of the C type `long`.

Errors:

It is an error if `vhpi_get_time` is called while the tool is any execution phase other than the initialization or simulation phases. It is an error if the `time` and `cycles` arguments are both `NULL`.

See also:

`vhpi_get_phys(vhpiResolutionLimitP, NULL)`, `vhpi_get_next_time`.

NOTE—A VHPI program can use the `vhpi_format_value` function to change the way in which the time value is expressed.

Example:

In the following VHPI program, the `vhpi_get_time` function is used to get the current simulation time without the count of delta cycles.

```
vhpiTimeT time;

vhpi_get_time(&time, NULL);
vhpi_printf("time = %d %d\n", time.high, time.low);
```

23.19 vhpi_get_value

Gets the formatted value of an object that has a value.

Synopsis:

```
int vhpi_get_value (vhpiHandleT expr, vhpiValueT *value_p);
```

Description:

The `expr` argument is a handle to an object of a class that has the `vhpi_get_value` operation. The `value_p` argument is a pointer to a value structure specifying the format and containing storage into which the formatted value is written. Storage for the value structure is allocated by the VHPI program before calling the function.

The function reads the value of the object referred to by `expr` (see 22.3) and represents it in the format specified in the `format` member of the value structure (see 22.2). If the formatted value is a scalar, the function writes the formatted value to the `value` member of the value structure. If the formatted value is represented as an array, string, or using internal representation and the value of the `value` member of the value structure is `NULL`, the function does not write the formatted value, but returns the minimum number of bytes of storage that would be required to write the value.

If the formatted value is represented as an array, string, or using internal representation, the VHPI program, before calling `vhpi_get_value`, may allocate storage for the formatted value and write the size in bytes and the address of the storage into the `bufSize` and `value` members, respectively, of the value structure. In that case, the function writes the formatted value to the storage pointed to by the `value` member of the value structure (see 22.2).

If the format specified in the `format` member of the value structure is `vhpiObjTypeVal`, the representation of the formatted value depends on the type of the object referred to by `objHdl` (see 22.4). The function writes to the `format` member of the value structure the value of type `vhpiFormatT` corresponding to the type.

If the formatted value is represented as a physical or time value or an array of physical or time values, the function writes to the `unit` member of the value structure the position number of a scale factor. If the object referred to by `expr` is a physical or time literal, the scale factor is the position number of the unit of the literal; otherwise, the scale factor is 1.

Return value:

0 if the formatted value is a scalar and the operation completes without error, or if the formatted value is represented as an array, string, or using internal representation, the `value` member of the value structure is not `NULL`, the size provided in the `bufSize` member of the value structure is sufficient and the operation

completes without error; or the minimum size in bytes of storage required to represent the value in the specified format if the formatted value is represented as an array, string, or using internal representation and either the `value` member of the value structure is `NULL` or the size provided in the `bufSize` member of the value structure is insufficient; or a negative integer otherwise.

Errors:

It is an error if the `vhpi_get_value` function is passed a handle that refers to a VHDL object for which reading is not permitted (see 6.5.2). In particular, it is an error if the `vhpi_get_value` function is passed a handle to an object that has the `Access` property and the value of that property does not have the `vhpiRead` flag set. It is an error if the `vhpi_get_value` function is passed a handle to an object of class `expr` that represents an expression that is not static.

It is an error if the `format` member of the value structure specifies a format that cannot be used to represent the value. It is an error if the formatted value is outside of the range of values that can be represented.

It is an error if the amount of storage allocated for a value represented as an array, string, or using internal representation is insufficient for the formatted value.

A tool may perform optimizations that make the value of an object inaccessible. It is an error if the handle `expr` refers to such an object.

See also:

`vhpi_put_value`, `vhpi_schedule_transaction`, `vhpi_format_value`.

23.20 vhpi_handle

Gets a handle to an object that is the target of a one-to-one association.

Synopsis:

```
vhpiHandleT vhpi_handle (vhpiOneToOneT type,
                         vhpiHandleT referenceHandle);
```

Description:

The `type` argument is an enumeration value that corresponds to a one-to-one association role. The `referenceHandle` argument is a handle to a reference object, that is, an object of the class that is the reference class of the one-to-one association.

If the association corresponding to the value of `type` has a multiplicity of 1, or if the association has a multiplicity of 0..1 and a target object is associated with the reference object, the function returns a handle to the target object of the association. If the association has a multiplicity of 0..1 and no object is associated with the reference object, the function returns `NULL`.

Return value:

A handle to the target object if one exists, or `NULL` otherwise.

Example:

In the following VHPI program, the function vhpi_handle is used to get handles to a parent region and design unit.

```
vhpiHandleT get_instance_info(vhpiHandleT scopeHdl) {
  vhpiHandleT upScopeHdl, duHdl;

  /* climb the hierarchy one level */
  /* traverse an association with an explicitly named role */
  upScopeHdl = vhpi_handle(vhpiUpperRegion, scopeHdl);
  if (vhpi_get(vhpiKindP, upScopeHdl) == vhpiCompInstStmtK) {
    /* traverse an association with an implicitly named role */
    duHdl = vhpi_handle(vhpiDesignUnit, upScopeHdl);
    return(duHdl);
  } else
    return(NULL);
} /* end get_instance_info() */
```

23.21 vhpi_handle_by_index

Gets a handle to an object that is a target of an ordered one-to-many association.

Synopsis:

```
vhpiHandleT vhpi_handle_by_index (vhpiOneToManyT itRel,
                                  vhpiHandleT parent, int32_t indx);
```

Description:

The itRel argument is an enumeration value that corresponds to an ordered one-to-many association role. The parent argument is a handle to a reference object, that is, an object of the class that is the reference class of the one-to-many association. The indx argument is the index of a target object in the one-to-many association.

If the one-to-many association has a number of target objects that is greater than the value of indx, the function returns a handle to the target object whose position in the set of target objects, starting from 0, is given by the value of indx; otherwise, the function returns NULL.

Return value:

A handle to the target object if one exists, or NULL otherwise.

See also:

vhpi_iterator, vhpi_scan.

NOTE 1—Those one-to-many associations that are ordered are specified as ordered associations in Clause 19.

NOTE 2—The result of calling vhpi_handle_by_index is equivalent to calling vhpi_iterator with the same first and second arguments, followed by indx + 1 successive calls to vhpi_scan applied to the resulting iterator.

Example:

In the following VHPI program, the vhpi_handle_by_index function is used to access the constraints of a given element of a composite object.


```
vhpiHandleT find_indexed_constraint(vhpiHandleT parentHdl, int index) {
  vhpiHandleT subtypeHdl, typeHdl, subHdl;

  subtypeHdl = vhpi_handle(vhpiType, parentHdl);
  typeHdl = vhpi_handle(vhpiBaseType, subtypeHdl);
  if (vhpi_get(vhpiIsCompositeP, typeHdl)) {
    /* get the given indexed array element or indexed record field
       of the parent object */
    subHdl = vhpi_handle_by_index(vhpiConstraints, parentHdl, index);
    return subHdl;
  }
  else
    return NULL;
}
```

In the following VHPI program, the `vhpi_handle_by_index` function is used to access the first formal parameter of a called subprogram. The formal parameter declarations associated with a subprogram call object are ordered according to the declaration of the parameters in the subprogram's interface list. A handle to the subprogram call object is acquired from a callback information structure.

```
void exec_proc(vhpiCbDataT cbDatap) {
  vhpiHandleT subpCallHdl, formal1, formalIt;
  int val = 0;
  vhpiValueT value;

  value.format = vhpiIntVal;
  value.value->integer = &val;
  subpCallHdl = cbDatap->obj;

  /* get a handle to the first formal parameter
     of the subprogram call */
  formal1 = vhpi_handle_by_index(vhpiParamDecls, subpCallHdl, 0);

  switch(vhpi_get(vhpiModeP, formal1)) {
  case vhpiIN:
    vhpi_get_value(formal1, &value);
    break;
  case vhpiOUT:
    vhpi_put_value(formal1, &value);
    break;
  default:
    break;
  }
}
```

Given the following VHDL declarations:

```
type my_1D_array is array (2 to 5) of bit;
type my_2D_array is array (2 to 5, 3 to 5) of integer;

variable A: my_1D_array := ('1', '0', '1', '0');
variable M: my_2D_array := ((1, 2, 3),
                            (4, 5, 6),
                            (7, 8, 9),
```

```
                                (10, 11, 12));

type myrecord is record
   I: integer;
   B: bit;
   AR: my_1D_array;
end record;
type myrecord_ptr is access myrecord;
type mybit_vector_ptr is access bit_vector;

variable R: myrecord := (9, '0', B"1111");
variable R_p: myrecord_ptr;
variable BV_p: mybit_vector_ptr;
```

the following statements in a VHPI program use the `vhpi_handle_by_index` function to access elements of the VHDL variables, as described by the comments:

```
/* if Ahdl is an handle to variable A, hdl is a handle to A(2) */
hdl = vhpi_handle_by_index(vhpiIndexedNames, Ahdl, 0)

/* if Mhdl is an handle to variable M, hdl is handle to M(2,3) */
hdl = vhpi_handle_by_index(vhpiIndexedNames, Mhdl, 0)

/* if Rhdl is an handle to variable R, hdl is a handle to R.I */
hdl = vhpi_handle_by_index(vhpiSelectedNames, Rhdl, 0)

/* if Rhdl is an handle to variable R, subeltHdl is a handle to R.AR */
subeltHdl = vhpi_handle_by_index(vhpiSelectedNames, Rhdl, 2)
/* and hdl is a handle to R.AR(4} */
hdl = vhpi_handle_by_index(vhpiIndexedNames, subeltHdl, 2)

/* if BV_phdl is an handle to variable BV_p,
   hdl is a handle to BV_p(0) */
hdl = vhpi_handle_by_index(vhpiIndexedNames, BV_phdl, 0)

/* if R_phdl is an handle to variable R_p, hdl is a handle to R_p.I */
hdl = vhpi_handle_by_index(vhpiSelectedNames, R_phdl, 0)
```

23.22 vhpi_handle_by_name

Gets a handle to an object that is identified by its name.

Synopsis:

```
vhpiHandleT vhpi_handle_by_name (const char *name, vhpiHandleT scope);
```

Description:

The `name` argument is a pointer to a string referred to as the *search string*. The `scope` argument is a handle to an object of class `region` that represents an instantiated declarative region in the design hierarchy information model; or a handle to an object of class `lexicalScope` that represents an uninstantiated scope in the library information model; or `NULL`. If the `scope` argument is not `NULL`, the object referred to by the handle is referred to as the *scope object*.

The function uses the search string to locate an object that has the FullName property and whose value for that property is matched by the search string. In determining whether a search string matches the value of a FullName property, letters are compared without regard to case, unless the letters occur in an extended identifier, in which case the case of letters is significant.

The search string may be of the form described in 19.4.6 for the value of the DefName property of an object in the library information model, except that for each occurrence of a subprogram name or enumeration literal within the search string, a signature may be inserted immediately following the subprogram name or enumeration literal. Such a search string is referred to as an *absolute library search string* and matches a FullName property that is the same string excluding any signatures. If the search string is an absolute library search string and the scope argument is not NULL, the scope object shall be of class lexicalScope. In that case, the vhpi_handle_by_name function limits the search to those objects representing named entities contained, directly or indirectly, in the declarative region represented by the scope object. Otherwise, if the scope argument is NULL, the vhpi_handle_by_name function searches in the entire library information model. In either case, for each signature in the search string, if any, the search is further limited to those objects representing named entities contained, directly or indirectly, in the declarative region represented by the object whose Name property matches the subprogram name or enumeration literal immediately preceding the signature and whose SignatureName property matches the signature.

The search string may be of the form described in 19.4.7 for the value of the FullName property of an object in the design hierarchy information model, except that for each occurrence of a subprogram name or enumeration literal within the search string, a signature may be inserted immediately following the subprogram name or enumeration literal. Such a search string is referred to as an *absolute design hierarchy search string*, and matches a FullName property that is the same string excluding any signatures. If the search string is an absolute design hierarchy search string and the scope argument is not NULL, the scope object shall be of class region or decl. In that case, the vhpi_handle_by_name function limits the search to those objects representing named entities contained, directly or indirectly, in the instantiated region or elaborated declaration, as appropriate, represented by the scope object. Otherwise, if the scope argument is NULL, the vhpi_handle_by_name function searches in the entire design hierarchy information model. In either case, for each signature in the search string, if any, the search is further limited to those objects representing named entities contained, directly or indirectly, in the instantiated region represented by the object whose Name property matches the subprogram name or enumeration literal immediately preceding the signature and whose SignatureName property matches the signature.

A search string in a form other than that of an absolute design hierarchy search string or an absolute library search string is referred to as a *relative search string*. If the search string is a relative search string and the scope argument is not NULL, the effect of the call to the vhpi_handle_by_name function is the same as that of a call to the function with the same scope argument and a *modified relative search string*, formed by concatenating the following two strings in the following order:

— The value of the FullName property of the scope object, into which is inserted, immediately after each occurrence of a subprogram name or enumeration literal, the value of the SignatureName property of the object representing the subprogram or enumeration literal denoted by the subprogram name or enumeration literal, and

— The relative search string.

A search that locates more than one object is ambiguous. The tool may detect that the search is ambiguous and return NULL. If the tool does not detect that the search is ambiguous, it returns a handle to one of the located objects chosen in an implementation-defined manner.

Return value:

A handle to a located object, if any, or NULL otherwise.

Errors:

It is an error if the search string is a relative search string and the scope argument is `NULL`.

It is an error if the search string is a relative search string and the modified relative search string is neither a well-formed absolute library search string nor a well-formed absolute design hierarchy search string.

See also:

`vhpi_get_str(vhpiNameP, ...)`, `vhpi_get_str(vhpiFullNameP, ...)`.

Example:

In the following VHPI program, the `vhpi_handle_by_name` function is used to search for a signal of a given simple name within a design hierarchy.

```
vhpiHandleT findsignal(char *sigName) {
  vhpiHandleT subitr, hdl, subhdl, sigHdl;
  /* first search for the signal in the design hierarchy, starting at
     the root instance level and recursively descending into the
     sub-instances
  */
  itr = vhpi_handle(vhpiRootInst, NULL);
  if (itr) {
    sigHdl = vhpi_handle_by_name(sigName, hdl);
    if (sigHdl)
      return sigHdl;
    else {
      subitr = vhpi_iterator(vhpiInternalRegions, hdl);
      if (subitr)
        while (subhdl = vhpi_scan(subitr)) {
          sigHdl = vhpi_handle_by_name(sigName, subhdl);
          if (sigHdl)
            return sigHdl;
        }
    }
  }
  /* if not found in the design hierarchy, search for the signal
     in the instantiated packages
  */
  itr = vhpi_iterator(vhpiPackInsts, NULL);
  if (itr)
    while (hdl = vhpi_scan(itr)) {
      sigHdl = vhpi_handle_by_name(sigName, hdl);
      if (sigHdl)
        return sigHdl;
    }
  return NULL;
}
```

23.23 vhpi_is_printable

Determines whether a given character is a graphic character.

Synopsis:

```
int vhpi_is_printable( char ch )
```

Description:

The function tests whether the character code that is the value of the `ch` argument represents a graphic character (see 15.2).

Return value:

1 if the character is a graphic character, or 0 otherwise.

23.24 vhpi_iterator

Creates an iterator for a one-to-many association.

Synopsis:

```
vhpiHandleT vhpi_iterator (vhpiOneToManyT type,
                           vhpiHandleT referenceHandle);
```

Description:

The `type` argument is an enumeration value that corresponds to a one-to-many association role. The `referenceHandle` argument is a handle to a reference object, that is, an object of the class that is the reference class of the one-to-many association.

If the one-to-many association has one or more target objects, the function creates a new object of class `iterator`, initializes the iterator set of the object to be the set of target objects in the one-to-many association, initializes the iteration position of the object to refer to the first element in the iterator set, and returns a handle that refers to the object of class `iterator`. Otherwise, the function returns `NULL`.

If the one-to-many association is ordered, the elements in the iterator set are ordered in the same order as the target objects of the one-to-many association to which they refer. Otherwise, the order of elements in the iterator set is not specified by this standard.

Return value:

A handle to the object of class `iterator`, if such an object is created, or `NULL` otherwise.

See also:

`vhpi_scan`.

NOTE—Since each call to the `vhpi_iterator` function creates a new object of class `iterator`, handles returned by separate calls to the function are distinct, and comparison of such handles using the `vhpi_compare_handles` function always yields `vhpiFalse`.

Example:

In the following VHPI program, the `vhpi_iterator` function is used to create an iterator for all signals in a scope.

```
void find_signals(vhpiHandleT scopeHdl) {
  vhpiHandleT sigHdl,itrHdl;

  /* find all signals in the scope and print their names */

  itrHdl = vhpi_iterator(vhpiSigDecl, scopeHdl);
  if (!itrHdl) return;
  while (sigHdl = vhpi_scan(itrHdl)) {
    vhpi_printf("Found signal %s\n", vhpi_get_str(vhpiNameP, sigHdl));
    vhpi_release_handle(sigHdl);
  }
}
```

23.25 vhpi_printf

Writes a message to one or more tool output files.

Synopsis:

```
int vhpi_printf (const char *format, ...);
```

Description:

The `format` argument is a pointer to a format string that may contain conversion codes as defined for the C `printf` function in ISO/IEC 9899:1999/Cor 1:2001. The format string and subsequent arguments to the `vhpi_printf` function are interpreted in the same way as specified in ISO/IEC 9899:1999/Cor 1:2001 for the C `printf` function to form a formatted character string that is written to one or more tool output files. The file or files to which the string is written is determined in an implementation-defined manner.

Return value:

The number of characters written to the file, or –1 if an error occurred.

See also:

`vhpi_is_printable`.

NOTE—The file or files to which `vhpi_printf` writes may include a standard output stream or a tool log file.

Example:

In the following VHPI program, the `vhpi_printf` function is used to print a character string with non-graphic characters represented using textual representations of the corresponding enumeration literal of the VHDL standard CHARACTER type.

```
int PrintMyNastyVHDLString( char* VHDLString, int Length ) {
  int i;
  unsigned char ch;
  int needcomma=0;
  for (i=0; i<Length; i++) {
    ch = (unsigned char)VHDLString[i];
    if (vhpi_is_printable(ch)) {
      vhpi_printf("%c", ch );
      needcomma=1;
```



```
    } else {
      if (needcomma)
        vhpi_printf(",");
      vhpi_printf("%s", VHPI_GET_PRINTABLE_STRINGCODE(ch));
      if (i!=(Length-1))
        vhpi_printf(",");
      needcomma=0;
    }
  }
  return 0;
}
```

A call to the function `PrintMyNastyVHDLString` with the string yielded by the following VHDL expression:

```
"HELLO" & NUL & C128 & DEL
```

would cause the following character string to be written to the file:

```
HELLO,NUL,C128,DEL
```

23.26 vhpi_protected_call

Calls a function to operate on a shared variable of a protected type.

Synopsis:

```
int vhpi_protected_call (vhpiHandleT varHdl,
                         vhpiUserFctT userFct, void *userData);
```

Description:

The `varHdl` argument is a handle to an object of class `varDecl` for which the properties `IsShared` and `IsProtectedType` both have the value `vhpiTrue`. The `userFct` argument is a pointer to a function to be called with exclusive access to the object referred to by `varHdl`. The `userData` argument is a pointer to be passed to the function pointed to by the `userFct` argument.

The `vhpi_protected_call` function blocks (suspends execution while retaining all state), if necessary, until exclusive access to the object referred to by `varHdl` is secured. The `vhpi_protected_call` function then calls the function pointed to by `userFct`. The first argument passed to the function is the value of the `varHdl` argument, and the second argument passed to the function is the value of the `userData` argument. Upon return of the function, exclusive access to the object referred to by `varHdl` is rescinded.

The function pointed to by the `userFct` argument is assumed to have the prototype

```
int userFct (vhpiHandleT varHdl, void *userData);
```

Return value:

The value returned by the function pointed to by the `userFct` argument.

Errors:

A VHPI program that performs a read or write access to a shared variable of a protected type other than from within a function invoked by a call to the `vhpi_protected_call` function with the first argument being a handle to the variable is erroneous.

NOTE 1—The effects of acquiring and rescinding exclusive access to a variable of protected type using the `vhpi_protected_call` function are equivalent to the effects of acquiring and rescinding exclusive access using calls to protected-type methods within a VHDL model (see 4.3 and 14.6).

NOTE 2—The value of the `userData` argument may be `NULL`.

Example:

In the following VHPI program, the `vhpi_protected_call` function is used to acquire exclusive access to a variable named `Foo`, which has a private variable named `result`. A pointer to the function `Myfunc` is passed to the `vhpi_protected_call` function. The function `Myfunc` reads the value of the `result` variable, invokes a function to perform an operation on the value, and writes a new value to the variable.

```
#define FAIL -1;
typedef struct { int Value;
                 int Size;
                 in Op;} MyData;

/* user function which is called on the protected variable handle */
int Myfunc( vhpiHandleT protectedVarDeclHdl, void* ClientData ) {
  int status=0;
  vhpiHandleT resultH;
  MyData* Data=(MyData*)ClientData;

  /* result is a private variable declaration for the protected type */
  resultH = vhpi_handle_by_name("result", protectedVarDeclHdl);
  if (!resultH)
    return(FAIL);

  /* access the current value of result */
  status = vhpi_get_value( resultH, Data->Value );
  if (status) {
    vhpi_printf("error in reading protected variable\n");
    return (status);
  }
  switch (Data->Op) {
  case op1:
    op1CB(Data->Value);
    break;
  case ...
  default:
    Bombout();
  }
  /* set result to a new value */
  status = vhpi_put_value( resultH, Data->Value, vhpiDeposit );
  /* do some more error checking */
  if (status)
    vhpi_printf("error in writing to protected variable\n");
  return status;
}
```



```
int op1CB( int value ) {
  ...
}

int main (int argc, char *argv[]) {
  /* get a handle to the protected variable declaration named "Foo" */
  vhpiHandleT protectedVarDeclHdl
    = vhpi_handle_by_name("Foo", vpi_handle(vhpiRootInst, NULL));
  MyData Data;
  int status = 0;

  Data.Op = op1;
  Data.Size = 100;
  bzero(Data.Value, Data.Size );

  if (protectedVarDeclHdl)
    status = vhpi_protected_call(protectedVarDeclHdl,Myfunc,Data);

  if (status)
    vhpi_printf("Unable to perform operation op1 "
                "with protected variable Foo\n");
  return(status);
}
```

23.27 vhpi_put_data

Saves data for restart.

Synopsis:

```
size_t vhpi_put_data (int32_t id,
                      void * dataLoc, size_t numBytes);
```

Description:

The `id` argument is an identification number for a saved data set. The `dataLoc` argument is the address from which data is read to be written to the saved data set. The `numBytes` argument is a positive number, being the number of bytes of data to write.

The function reads a number of bytes, given by `numBytes`, from the address pointed to by `dataLoc` and writes the data to the saved data set identified by `id`.

The first call to `vhpi_put_data` with a given value for `id` during a given occurrence of the save phase of tool execution writes bytes to the saved data set starting at the first location of the saved data set. Subsequent calls to `vhpi_put_data` with the same `id` value during the same occurrence of the save phase write bytes starting at the location immediately after the last location written by the immediately preceding call with the given `id` value.

A tool shall allow VHPI programs to call `vhpi_put_data` an unbounded number of times with a given identification number and with an unbounded number of different identification numbers, subject to resource constraints of the host system. The order in which sequences of calls to `vhpi_put_data` with different identification numbers are interleaved is not significant.

Return value:

The number of bytes actually written, or 0 if the write failed.

Errors:

It is an error if `vhpi_put_data` is called other than from a `vhpiCbStartOfSave` or `vhpiCbEndOfSave` callback.

It is an error if the `id` value is not valid for the occurrence of the save phase of tool execution during which the `vhpi_put_data` function is called.

See also:

`vhpi_get_data`.

NOTE—A VHPI program can acquire an identification number with the function call `vhpi_get(vhpiIdP, NULL)`. Each call of this form returns a unique non-zero identification number.

Example:

In the following VHPI program, the `vhpi_put_data` function is used first to write the number of linked list elements in a saved data set and second to write that number of linked list elements. The VHPI program function that calls `vhpi_put_data` is a `vhpiCbEndOfSave` callback. It registers a `vhpiCbStartOfRestart` callback to retrieve the data upon restart (see example in 23.12).

```
/* type definitions for private data structures to save used by the
   foreign models or applications */
struct myStruct{
  struct myStruct *next;
  int d1;
  int d2;
}

void consumer_save(vhpiCbDataT *cbDatap) {
  char *data;
  vhpiCbDataT cbData; /* a cbData structure */
  int cnt = 0;
  struct myStruct *wrk;
  vhpiHandleT cbHdl; /* a callback handle */
  int id =0;
  int savedBytesCount = 0;

  /* get the number of structures */
  wrk = firstWrk;
  while (wrk) {
    cnt++;
    wrk = wrk->next;
  }
  /* request an id */
  id = vhpi_get(vhpiIdP, NULL);
  /* save the number of data structures */
  savedBytesCount = vhpi_put_data(id, (char*)&cnt, sizeof(int));
  /* reinitialize wrk pointer to point to the first structure */
  wrk = firstWrk;
```



```
  /* save the different data structures, the restart routine will have
     to fix the pointers */
  while (wrk) {
    savedBytesCount += vhpi_put_data(id, (char *)wrk,
                                     sizeof(struct myStruct));
    wrk = wrk->next;
  }
  /* check if everything has been saved */
  assert(savedBytesCount == sizeof(int)
                          + cnt * (sizeof(struct myStruct)));
  /* now register the callback for restart and pass the id to retrieve
     the data, the user_data member of the callback data structure is
     one easy way to pass the id to the restart operation */
  cbData.user_data = (void *)id;
  cbData.reason = vhpiCbStartOfRestart;
  cbData.cb_rtn = consumer_restart;
  vhpi_register_cb(&cbData, vhpiNoReturn);
} /* end of consumer_save */
```

23.28 vhpi_put_value

Updates the value of an object or provides the return value of a foreign function call.

Synopsis:

```
int vhpi_put_value (vhpiHandleT object,
                    vhpiValueT *value_p, vhpiPutValueModeT mode);
```

Description:

The `object` argument is a handle to an object of class `objDecl`, `name`, or `driver`, or a handle to an object of class `funcCall` for which the associated `subpBody` object has the value `vhpiTrue` for the `IsForeign` property. The `value_p` argument is a pointer to a value structure, if required, specifying the value to be used to update the object or the return value of the foreign function call. The `mode` argument specifies how the update of the object is to be performed. The function updates the object value according to the rules of 22.5

Return value:

0 if the operation completes without error, or a non-zero value otherwise.

Errors:

It is an error if the `vhpi_put_value` function is passed a handle that refers to a VHDL object for which updating is not permitted (see 6.4.2.2, 6.4.2.5, and 6.5.2). In particular, it is an error if the `vhpi_put_value` function is passed a handle to an object that has the `Access` property and the value of that property does not have the `vhpiWrite` flag set.

It is an error if the `vhpi_put_value` function is called with an update mode of `vhpiForcePropagate` or `vhpiDepositPropagate` to update a member of a resolved composite signal.

It is an error if the vhpi_put_value function is called with an update mode of vhpiForcePropagate or vhpiDepositPropagate to update a member of a resolved composite signal.

It is an error if the vhpi_put_value function is called during substep 6) of step h) of the simulation cycle to cause activity on a driver or a signal (see 14.7.5.3 and 21.3.6.8).

See also:

vhpi_get_value, vhpi_schedule_transaction.

NOTE—A VHPI program shall not use a format for which not all values of the object's type can be represented (see 22.2), even if the value with which the object is to be updated can be represented using that format. For example, it would be an error to update an object of a physical type whose position numbers exceeded the bounds of 32-bit representation using the vhpiSmallPhysVal format.

23.29 vhpi_register_cb

Registers a callback.

Synopsis:

```
vhpiHandleT vhpi_register_cb (vhpiCbDataT *cb_data_p, int32_t flags);
```

Description:

The cb_data_p argument is a pointer to a callback data structure. The flags argument is a value that specifies how the callback is to be registered. The function uses the information in the callback data structure to register a callback function according to the rules of Clause 21.

Annex B defines two callback flags, vhpiReturnCb and vhpiDisableCb. A call to the function is said to set a callback flag if the value of the flags argument has a 1 bit at the bit position corresponding to the 1 bit in the value of the callback flag; otherwise the call to the function is said to clear the callback flag.

If a call to the function sets the vhpiReturnCb flag, the function returns a handle to an object of class callback that represents the registered callback. If a call to the function clears the vhpiReturnCb flag, the function returns NULL.

If a call to the function sets the vhpiDisableCb flag, the function sets the registered callback to the disabled state. If a call to the function clears the vhpiDisableCb flag, the function sets the registered callback to the enabled state.

Upon completion of the vhpi_register_cb function, the tool does not retain references to the storage pointed to by the cb_data_p argument or to storage pointed to by pointers within the callback data structure. Furthermore, if the obj member of the callback data structure contains a handle, the VHPI program may release the handle after the vhpi_register_cb function returns without affecting registration of the callback.

Return value:

A handle that refers to the registered callback, or NULL.

See also:

`vhpi_get_cb_info`, `vhpi_remove_cb`, `vhpi_enable_cb`, `vhpi_disable_cb`.

NOTE—A VHPI program that registers a callback with the `vhpiDisableCb` flag set may find it useful also to set the `vhpiReturnCb` flag and to save the returned handle. The program can subsequently use the handle to enable the callback without having to navigate associations to acquire a handle to the callback.

Example:

In the following VHPI program, the `vhpi_register_cb` function is used to register a value change callback for each signal within a component instance.

```
/* the callback function */
void vcl_trigger(const vhpiCbDataT *cbDatap) {
  char *sigName;
  int toggleCount = (int)(cbDatap->user_data);

  cbDatap->user_data = (char *)(++toggleCount);
  sigName= vhpi_get_str(vhpiFullNameP, cbDatap->obj);
  vhpi_printf("Signal %s changed value %d, at time %d\n",
              sigName, cbDatap->value.int, cbDatap->time.low);
  return;
}

/* this is the name of the function which registers signal
   value change callbacks to monitor all signals in an instance*/
static void monitorSignals(vhpiHandleT instHdl) {
  static vhpiCbDataT cbData;
  vhpiValueT value;
  vhpiTimeT time;
  int flags;

  value.format = vhpiIntVal;
  cbData.reason = vhpiCbValueChange;
  cbData.cb_rtn = vcl_trigger;
  cbData.value = &value;
  cbData.time = &time;
  cbData.user_data = 0;
  flags = 0; /* do not return a callback handle and do not disable
                the callback at registration */
  /* register the callback function */
  sigIt = vhpi_iterator(vhpiSigDecls, instHdl);
  if(!sigIt) return;
  while(sigHdl = vhpi_scan(sigIt)) {
    cbData.obj = sigHdl;
    vhpi_register_cb(&cbData, flags);
  }
}
```

23.30 vhpi_register_foreignf

Registers a foreign model or application.

Synopsis:

```
vhpiHandleT vhpi_register_foreignf (vhpiForeignDataT *foreignDatap);
```

Description:

The `foreignDatap` argument is a pointer to a foreign data structure. The function registers a foreign model or application according to the rules of (20.2) using the information in the foreign data structure.

The value of the `kind` member shall be the value of an enumeration constant of type `vhpiForeignKindT` defined in Annex B and identifies whether a foreign architecture, function, procedure, or application is registered. For registration of a foreign architecture, the value of the `elabf` member shall be a pointer to the elaboration function, if required, or `NULL` otherwise; and the value of the `execf` member shall be a pointer to the execution function. For registration of a foreign procedure or function, the value of the `elabf` member shall be `NULL` and the value of the `execf` member shall be a pointer to the execution function.

The value of the `libraryName` member shall be a pointer to a string whose value is the object library name. For registration of a foreign model or application, the value of the `modelName` member shall be a pointer to a string whose value is the model name or application name, respectively.

Return value:

A handle that refers to an object of class `foreignf` that represents the foreign model or application, if the operation completes without error; or `NULL` otherwise.

Errors:

It is an error of the `vhpi_register_foreignf` function is called other than during the registration phase of tool execution.

It is an error if the value of the `kind` member of the foreign data structure is `vhpiLibF`.

See also:

`vhpi_get_foreignf_info`, `vhpi_iterator(vhpiForeignfs, NULL)`.

Example:

In the following VHPI program, the `vhpi_register_foreignf` function is used to register dynamically linked elaboration and execution functions for a foreign model.

```
void dynlink(char * foreignName, char * libName) {
  /* foreignName is the name of the foreign model to link in */
  /* libName is the logical name of the C dynamic library where the
     model resides */
  static vhpiForeignDataT archData = {vhpiArchF};
  char dynLibName[MAX_STR_LENGTH];
  char platform[6];
  char extension[3];
  char fname[MAX_STR_LENGTH];
  char elabfname[MAX_STR_LENGTH];
  char execfname[MAX_STR_LENGTH];
```



```
  sprintf(platform, getenv("SYSTYPE"));
  if (!strcmp(platform, "SUNOS"))
    strcpy(extension, "so");
  else if (!strcmp(platform, "HP-UX"))
    strcpy(extension, "sl");

  sprintf(dynLibName, "%s.%s", libName, extension);
  sprintf(fname, "%s", foreignName);
  sprintf(elabfname, "elab_%s", foreignName);
  sprintf(execfname, "sim_%s", foreignName);
  archData->libraryName = libname;
  archData->modelName = fName;
  /* find the function pointer addresses */
  archData->elabf = (void(*)()) dynlookup(dynLibName, elabfName);
  archData->execf = (void(*)()) dynlookup(dynLibName, execfName);

  vhpi_register_foreignf(&archData);
}
```

In the following VHPI program, the `vhpi_register_foreignf` function is used to register each foreign model contained in a C library.

```
extern void register_my_C_models();
       /* this is the name of the bootstrap
          function that shall be the ONLY
          visible symbol of the C library.
       */

void register_my_C_models() {
  static vhpiForeignDataT foreignDataArray[] = {
    {vhpiArchF, "lib1", "C_AND_gate", "elab_and", "sim_and"},
    {vhpiFuncF, "lib1", "addbits", 0, "ADD"},
    {vhpiProcF, "lib1", "verify", 0, "verify"},
    0
  };
  /* start by the first entry in the array of
     the foreign data structures */
  vhpiForeignDatap foreignDatap = &(foreignDataArray[0]);
  /* iterate and register every entry in the table */
  while (*foreignDatap)
    vhpi_register_foreignf(foreignDatap++);
}
```

23.31 vhpi_release_handle

Releases a handle.

Synopsis:

```
int vhpi_release_handle (vhpiHandleT object);
```

Description:

The `object` argument is a handle that refers to an object. The function releases the handle (see 17.4).

Return value:

0 if the operation completes without error, or 1 otherwise.

Example:

In the following VHPI program, the `vhpi_release_handle` function is used to release each handle, returned by the `vhpi_scan` function applied to an iterator, up to but excluding the first handle that refers to an object of class `blockStmt`.

```
vhpiHandleT rootHdl, itrHdl;

rootHdl = vhpi_handle(vhpiRootInst, null);
itrHdl = vhpi_iterator(vhpiInternalRegions, rootHdl);
if (itrHdl) {
  while (instHdl = vhpi_scan(itrHdl)) {
    if (vhpi_get(vhpiKindP, instHdl) == vhpiBlockStmtK)
      break;
    /* free this instance handle */
    vhpi_release_handle(instHdl);
  }
}
```

23.32 vhpi_remove_cb

Removes a previously registered callback.

Synopsis:

```
int vhpi_remove_cb (vhpiHandleT cb_obj);
```

Description:

The `cb_obj` argument is a handle to a registered callback. The function removes the callback. Upon return, the handle is invalid.

Return value:

0 if the operation completes without error, or 1 otherwise.

See also:

`vhpi_register_cb`, `vhpi_get_cb_info`, `vhpi_enable_cb`, `vhpi_disable_cb`.

23.33 vhpi_scan

Gets a handle to an object in an iterator and advances the iterator.

Synopsis:

```
vhpiHandleT vhpi_scan (vhpiHandleT iterator);
```


Description:

The `iterator` argument is a handle that refers to an iterator object of class `iterator`. If the iteration position of the iterator object refers to no element of the iterator set of the iterator object, the function releases the handle that is the value of the `iterator` argument and returns `NULL`. Otherwise, the function returns a handle to that element referred to by the iterator position of the iterator object and updates the iterator position to refer to the subsequent element in the iterator set, if any, or to no object otherwise.

Return value:

A handle to an object of the iterator set, or `NULL`.

See also:

`vhpi_iterator`.

NOTE—If a VHPI program no longer requires an iterator that is not exhausted, the program should release the handle that refers to the iterator so that the tool may reclaim memory resources allocated for the iterator.

Example:

In the following VHPI program, the `vhpi_scan` function is used to acquire handles to successive signals within a given scope.

```
vhpiHandleT find_signals(vhpiHandleT scopeHdl) {
  vhpiHandleT sigHdl,itrHdl;
  int found = 0;

  itrHdl = vhpi_iterator(vhpiSigDecl, scopeHdl);
  if (!itrHdl) return;
  while (sigHdl = vhpi_scan(itrHdl)) {
    vhpi_printf("Found signal %s\n", vhpi_get_str(vhpiNameP, sigHdl));
    /* done with handle */;
    vhpi_release_handle(sigHdl);
  }
}
```

23.34 vhpi_schedule_transaction

Schedules a transaction on a driver or transactions on a collection of drivers.

Synopsis:

```
int vhpi_schedule_transaction (vhpiHandleT  drivHdl,
                               vhpiValueT  *value_p,
                               uint32_t    numValues,
                               vhpiTimeT   *delayp,
                               vhpiDelayModeT delayMode,
                               vhpiTimeT    *pulseRejp);
```

Description:

The `drivHdl` argument is a handle that refers to an object of class `driver` or `driverCollection`. The `value_p` argument is a pointer to a value structure or to an array of value structures, or `NULL`. The

`numValues` argument is the number of value structures. The function schedules a transaction or transactions on the driver or drivers referred to by the `drivHDL` argument using the value or values specified by the `value_p` and `numValues` arguments, according to the rules of 22.6.

The `delayp` argument is a pointer to a time structure that specifies the relative delay. The time component of the transaction or transactions scheduled by the function is the sum of the current simulation time and the relative delay. If the value is less than the resolution limit of the tool, the transaction or transactions are scheduled with zero delay.

The `delayMode` argument is an enumeration constant that specifies the delay mechanism. The value of the `delayMode` argument shall be one of `vhpiInertial`, in which case the delay is construed to be inertial delay, or `vhpiTransport`, in which case the delay is construed to be transport delay (see 10.5.2.1).

The `pulseRejp` argument is a pointer to a time structure that specifies the pulse rejection limit or `NULL`.

If the `delayMode` argument is `vhpiInertial` and the `pulseRejp` argument is not `NULL`, the value of the time structure pointed to by the `pulseRejp` argument is the pulse rejection limit. The value shall not be greater than the delay. If the `delayMode` is `vhpiInertial` and the `pulseRejp` argument is `NULL`, the pulse rejection limit is equal to the delay. If the `delayMode` argument is `vhpiTransport`, the `pulseRejp` argument is ignored by the tool.

Return value:

0 if the operation completes without error, or a non-zero value otherwise.

Errors:

It is an error if the `vhpi_schedule_transaction` function is called other than during step f) of the simulation cycle or to schedule a transaction with non-zero delay during substeps 1) through 4) of step h) of the simulation cycle (see 14.7.5.3).

It is an error if the `vhpi_schedule_transaction` function is passed a handle to an object of class `driver` for which the `Access` property does not have the `vhpiWrite` flag set. Similarly, it is an error if the `vhpi_schedule_transaction` function is passed a handle to an object of class `driverCollection` and there is a member of the collection represented by the object for which the `Access` property does not have the `vhpiWrite` flag set.

See also:

`vhpi_put_value`, `vhpi_get_value`.

NOTE 1—An object of class driver is associated with a basic signal, which cannot be a composite non-resolved signal. To schedule a transaction for a composite non-resolved signal, a VHPI program may either schedule transactions individually for the driver of each of the subelements or may schedule a transaction on a collection comprising the drivers of the subelements.

NOTE 2—A VHPI program shall not use a format for which not all values of the type of the driver's signal can be represented (see 22.2), even if the value of the transaction can be represented using that format. For example, it would be an error to schedule a transaction on a driver for a signal of an integer type whose position numbers exceeded the bounds of 32-bit representation using the `vhpiIntVal` format.

Example:

In the following recursive VHPI program, the `vhpi_schedule_transaction` function is used to schedule transactions with the value '0' on each driver for each basic-signal subelement of type BIT of a signal. Handles to individual driver elements are acquired using iterators.


```
int schedule_transaction_value(vhpiHandleT sigHdl) {
  vhpiHandleT baseTypeHdl, driverIt, driverHdl;
  char *name;
  vhpiValueS value;
  vhpiTimeS delay;

  delay.low = 1000;/* delay is 1 ns */
  delay.high = 0;

  baseTypeHdl = vhpi_handle(vhpiBaseType, sigHdl);

  /* check the signal type */
  switch (vhpi_get(vhpiKindP, baseTypeHdl)) {
  case vhpiRecordTypeDeclK :
    {
      vhpiHandleT itsel, selh;
      if (!vhpi_get(vhpiIsResolved, sigHdl)) {
        /* signal not resolved at the composite level */
        itsel = vhpi_iterator(vhpiSelectedNames, sigHdl);
        while (selh = vhpi_scan(itsel))
          schedule_transaction_value(selh);
      } else {
        vhpi_printf("unimplemented\n");
        return -1;
      }
    }
    break;

  case vhpiArrayTypeDeclK:
    { /* get the element subtype */
      vhpiHandleT eltSubtypeHdl, bitIt, bitHdl;
      vhpiHandleT colHdl = NULL;
      int countdrivs = 0;
      if (vhpi_get(vhpiIsResolved, sigHdl)) {
        vhpi_printf("unimplemented\n");
        return -1;
      }
      /* signal not resolved at the composite level */
      elemSubtypeHdl = vhpi_handle(vhpiElemType, baseTypeDecl);
      baseTypeHdl = vhpiHandle(vhpiBaseType, elemSubtypeHdl);
      name = vhpi_get_str(vhpiNameP, baseTypeHdl);
      if (!strncmp(name, "BIT")) {
        bitIt = vhpi_iterator(vhpiIndexedNames, sigHdl);
        while (bitHdl = vhpi_scan(bitIt)) {
          assert (vhpi_get(vhpiIsBasicP, bitHdl) == vhpiTrue);
          driverIt = vhpi_iterator(vhpiDrivers, bitHdl);
          while (driverHdl = vhpi_scan(driverIt)) {
            countdrivs++;
            colHdl = vhpi_create(vhpiDriverCollectionK,
                                 colHdl, driverHdl);
          }
        }
        value.format = vhpiLogicVecVal;
        value.numElems = countDrivs;
```

```
        while (countdrivs) {
          value.value.logics++ = vhpiBit0;
          countdrivs--;
        }
        vhpi_schedule_transaction(colHdl, &value, 1,
                                  &delay, vhpiInertial, 0);
      } else {
        vhpi_printf("unimplemented\n");
        return -1;
      }
    }
    break;

  case vhpiEnumTypeDeclK:
    {
      name = vhpi_get_str(vhpiNameP, baseTypeHdl);
      if (!strncmp(name, "BIT")) {
        value.format = vhpiLogicVal;
        value.logic = vhpiBit0;
        assert (vhpi_get(vhpiIsBasicP, sigHdl) == vhpiTrue);
        driverIt = vhpi_iterator(vhpiDrivers, sigHdl);
        while (driverHdl = vhpi_scan(driverIt))
          countdrivs++;
        assert (countDrivs == 1);
        vhpi_schedule_transaction(driverHdl, &value, 1,
                                  &delay, vhpiInertial, 0);
      } else {
        vhpi_printf("unimplemented\n");
        return -1;
      }
    }
    break;

  default:
    vhpi_printf("unimplemented\n");
    return (-1);
    break;
  }
}
```

The VHPI program could be used to schedule transactions on subelements of a VHDL signal declared as follows:

```
type R is record
   B: BIT;
   BARR: BIT_VECTOR (0 to 7);
end record;

signal S: R;
```

23.35 vhpi_vprintf

Writes a message to one or more tool output files.

Synopsis:

```
int vhpi_vprintf (const char *format, va_list args);
```

Description:

The `format` argument is a pointer to a format string that may contain conversion codes as defined for the C `vprintf` function in ISO/IEC 9899:1999/Cor 1:2001. The format string and the `va_list` argument to the `vhpi_vprintf` function are interpreted in the same way as specified in ISO/IEC 9899:1999/Cor 1:2001 for the C `vprintf` function to form a formatted character string that is written to one or more tool output files. The file or files to which the string is written is determined in an implementation-defined manner.

Return value:

The number of characters written to the file, or –1 if an error occurred.

See also:

`vhpi_is_printable`.

NOTE—The file or files to which `vhpi_vprintf` writes may include a standard output stream or a tool log file.

24. Standard tool directives

24.1 Protect tool directives

24.1.1 General

Protect tool directives[14] allow exchange of VHDL descriptions in which portions are encrypted. This allows an author of a VHDL description to provide the description to one or more users in such a way that the users' tools can process the description, but the text of the description is not disclosed to the users.

A protect directive is a tool directive in which the identifier is the word **protect**. A protect directive directs the tool to perform encryption or decryption of a portion of the text of a VHDL design file. Protect directives are used to form *protection envelopes*, which include specification of cryptographic methods and keys to be used by a tool. An *encryption envelope* contains protect directives and a portion of the description, called the *source text*, that is to be encrypted. A *decryption envelope* contains protect directives and previously encrypted text to be decrypted.

Protection envelopes permit encryption and decryption of portions of descriptions using symmetric and asymmetric *ciphers*. A *symmetric cipher* involves use of the same key, called the *secret key*, for both encryption and decryption. An *asymmetric cipher* involves use of a *public key* for encryption and a corresponding *private key* for decryption.

Protection envelopes also permit encryption using *digital envelopes*, in which a portion of a description is encrypted using a symmetric cipher with an automatically generated *session key*, and then the session key is encrypted. Decryption of the protected envelope involves first decrypting the session key, followed by decrypting the portion of the description with the symmetric cipher using the decrypted session key.

The encrypted portion of a description may also be digitally signed by an author to allow checking that the encrypted text is unaltered. This involves computation of a *digest* by application of a *hash function* to the unencrypted text. The digest is then encrypted using an asymmetric cipher with the private key of the author. The decryption tool decrypts the description and recomputes the digest on the decrypted text. The decryption also decrypts the encrypted digest using the author's public key and compares the two digests. If they are the same, the description is unaltered; otherwise, it has been altered and should not be trusted.

Encrypted text, keys, and digests are *encoded* in decryption envelopes. An *encoding method* transforms the octets of encrypted information into graphic characters so that the information can be stored or transmitted without being altered by agents that interpret non-graphic characters.

An encryption envelope may contain a decryption envelope that is to be further encrypted. The result of encrypting the encryption envelope is a decryption envelope that contains an encrypted decryption envelope nested within it. The depth to which such decryption envelopes may be nested is implementation defined, but shall be no less than eight (that is, an innermost decryption envelope enclosed recursively within seven nested decryption envelopes).

The operation of encrypting an encryption envelope involves creating a corresponding decryption envelope as described in this subclause (24.1). The operation of decrypting a decryption envelope involves recreating the source text of the encryption envelope from which the decryption envelope was created.

[14]Material derived from the document titled "A Mechanism for VHDL Source Protection" © 2004, Cadence Design Systems Inc. Used, modified, and reprinted by permission.

As part of the analysis phase of tool execution (see Clause 20), a tool may perform encryption or decryption of a design file. The means by which it is determined whether the tool performs such processing is implementation defined.

It is an error if a protect tool directive appears other than as part of an encryption envelope or a decryption envelope. The effect of a protect tool directive, other than a protect decrypt license directive or a protect runtime license directive, is limited to the immediately enclosing protection envelope.

This standard does not specify any means by which encryption keys are exchanged among authors, users, and tools. It is assumed that the tools performing encryption and decryption have access to the required keys specified in protection envelopes. It is an error if a protection envelope requires use of a specified key and a tool processing the protection envelope does not have access to the key. Similarly, it is an error if a protection envelope requires use of one or more of a set of keys and a tool processing the protection envelope does not have access to any of the keys.

The graphic characters in a protect directive form a sequence of lexical elements that conform to the following grammar:

protect_directive ::=
    `**protect** keyword_expression { , keyword_expression }

keyword_expression ::=
      keyword
    | keyword = literal
    | keyword = keyword_list

keyword_list ::= (keyword_expression { , keyword_expression })

keyword ::= identifier

A protect directive containing more than one keyword expression is equivalent to a sequence of protect directives, each containing one keyword expression. The protect directives appear in the sequence in the same order as the keyword expressions in the original protect directive.

The directive identifier **protect** and the various keywords defined for each protect directive are shown in boldface in this subclause (24.1). The individual protect directives are described in 24.1.2, and the literals used to identify ciphers, hash functions, and encodings are described in 24.1.3. Rules for forming and processing encryption and decryption envelopes are described in 24.1.4 and 24.1.5, respectively.

Example:

The protect directive

```
`protect data_keyowner="ACME Corp.", data_keyname="secret-1",
data_method="aes192-cbc"
```

is equivalent to the following sequence of protect directives:

```
`protect data_keyowner="ACME Corp."
`protect data_keyname="secret-1"
`protect data_method="aes192-cbc"
```

NOTE—Products that include cryptographic algorithms may be subject to government regulations in some jurisdictions. Users of this standard are advised to seek the advice of competent counsel to determine their obligations under those regulations.

24.1.2 Protect directives

24.1.2.1 Protect begin directive

protect_begin_directive :: = `**protect begin**

A protect begin directive is part of an encryption envelope and indicates the beginning of the text of a description to be encrypted. The text to be encrypted, if any, begins with the first character after the end of the line containing the protect begin directive and ends with the character immediately preceding the next protect end directive.

24.1.2.2 Protect end directive

protect_end_directive :: = ` **protect end**

A protect end directive is part of an encryption envelope and indicates the end of the text of a description to be encrypted.

24.1.2.3 Protect begin protected directive

protect_begin_protected_directive :: = ` **protect begin_protected**

A protect begin protected directive forms the beginning of a decryption envelope.

24.1.2.4 Protect end protected directive

protect_end_protected_directive :: = ` **protect end_protected**

A protect end protected directive forms the end of a decryption envelope.

24.1.2.5 Protect author directive

protect_author_directive :: =
    ` **protect author** = string_literal

A protect author directive identifies the author of the portion of the VHDL description in the enclosing encryption or decryption envelope. The string literal identifies the author.

If a protect author directive appears in an encryption envelope, other than in the source text, then the directive shall appear unchanged in the corresponding decryption envelope. If a protect author directive appears in a decryption envelope, it has no effect on decryption of the decryption envelope.

24.1.2.6 Protect author info directive

protect_author_info_directive :: =
    ` **protect author_info** = string_literal

A protect author info directive provides descriptive information about the author of the portion of the VHDL description in the enclosing encryption or decryption envelope. The string literal provides the descriptive information.

If a protect author info directive appears in an encryption envelope, other than in the source text, then the directive shall appear unchanged in the corresponding decryption envelope. If a protect author info directive appears in a decryption envelope, it has no effect on decryption of the decryption envelope.

24.1.2.7 Protect encrypt agent directive

protect_encrypt_agent_directive :: =
` **protect encrypt_agent** = string_literal

A protect encrypt agent directive identifies the tool that created the enclosing decryption envelope. The string literal identifies the tool. An encryption tool shall include a protect encrypt agent directive in each decryption envelope it creates. The directive has no effect on decryption of the decryption envelope.

24.1.2.8 Protect encrypt agent info directive

protect_encrypt_agent_info_directive :: =
` **protect encrypt_agent_info** = string_literal

A protect encrypt agent info directive provides descriptive information about the tool that created the enclosing decryption envelope. The string literal provides the descriptive information. The directive has no effect on decryption of the decryption envelope.

24.1.2.9 Protect key keyowner directive

protect_key_keyowner_directive :: = ` **protect key_keyowner** = string_literal

A protect key keyowner directive identifies the owner of a key or key pair used to encrypt a session key. The string literal identifies the person, organization, or tool that owns the key or key pair.

24.1.2.10 Protect key keyname directive

protect_key_keyname_directive :: = ` **protect key_keyname** = string_literal

A protect key keyname directive identifies a particular key or key pair of a given key owner used to encrypt a session key. The string literal is the name of the key or key pair. If a key owner has more than one key, the key name may be used jointly with the key owner identified in a protect key keyowner directive to identify a given key or key pair.

24.1.2.11 Protect key method directive

protect_key_method_directive :: = ` **protect key_method** = string_literal

A protect key method directive identifies the cipher used to encrypt a session key. The string literal identifies the cipher (see 24.1.3.2).

24.1.2.12 Protect key block directive

protect_key_block_directive :: = ` **protect key_block**

A protect key block directive specifies use of a digital envelope. A protect key block directive appearing in an encryption envelope specifies that the encryption tool shall generate a session key to encrypt the portion of the VHDL description in the encryption envelope, and that the session key be encrypted. The corresponding decryption envelope shall contain a corresponding key block containing the encrypted session key. A protect key block directive appearing in a decryption envelope indicates that an encrypted session key immediately follows.

24.1.2.13 Protect data keyowner directive

protect_data_keyowner_directive :: = ` **protect data_keyowner** = string_literal

A protect data keyowner directive identifies the owner of a key or key pair used to encrypt a portion of a VHDL description. The string literal identifies the person, organization, or tool that owns the key or key pair.

24.1.2.14 Protect data keyname directive

protect_data_keyname_directive :: = ` **protect data_keyname** = string_literal

A protect data keyname directive identifies a particular key or key pair of a given key owner used to encrypt a portion of a VHDL description. The string literal is the name of the key or key pair. If a key owner has more than one key, the key name may be used jointly with the key owner identified in a protect data keyowner directive to identify a given key or key pair.

24.1.2.15 Protect data method directive

protect_data_method_directive :: = ` **protect data_method** = string_literal

A protect data method directive identifies the cipher used to encrypt a portion of a VHDL description. The string literal identifies the cipher (see 24.1.3.2).

24.1.2.16 Protect data block directive

protect_data_block_directive :: = ` **protect data_block**

A protect data block directive appearing in a decryption envelope indicates that an encrypted portion of a VHDL description immediately follows.

24.1.2.17 Protect digest keyowner directive

protect_digest_keyowner_directive :: = ` **protect digest_keyowner** = string_literal

A protect digest keyowner directive identifies the owner of a key pair used to encrypt a digest of a portion of a VHDL description. The string literal identifies the person, organization, or tool that owns the key pair.

24.1.2.18 Protect digest keyname directive

protect_digest_keyname_directive :: = ` **protect digest_keyname** = string_literal

A protect digest keyname directive identifies a particular key pair of a given key owner used to encrypt a digest of a portion of a VHDL description. The string literal is the name of the key pair. If a key owner has more than one key, the key name may be used jointly with the key owner identified in a protect digest keyowner directive to identify a given key pair.

24.1.2.19 Protect digest key method directive

protect_digest_key_method_directive :: = ` **protect digest_key_method** = string_literal

A protect digest key method directive identifies the cipher used to encrypt a digest of a portion of a VHDL description. The string literal identifies the cipher (see 24.1.3.2).

24.1.2.20 Protect digest method directive

protect_digest_method_directive :: = ` **protect digest_method** = string_literal

A protect digest method directive identifies a hash function used to compute a digest of a portion of a VHDL description. The string literal identifies the hash function (see 24.1.3.3).

24.1.2.21 Protect digest block directive

protect_digest_block_directive :: = ` **protect digest_block**

A protect digest block directive specifies use of a digital signature. A protect digest block directive appearing in an encryption envelope specifies that the encryption tool shall compute a digest of the portion of the VHDL description in the encryption envelope, and that the digest be encrypted. The corresponding decryption envelope shall contain a corresponding digest block containing the encrypted digest. A protect digest block directive appearing in a decryption envelope indicates that an encrypted digest immediately follows.

24.1.2.22 Protect encoding directive

protect_encoding_directive :: =
` **protect encoding** = (encoding_type_description
[, encoding_line_length_description] [, encoding_bytes_description])

encoding_type_description ::= **enctype** = string_literal

encoding_line_length_description ::= **line_length** = integer

encoding_bytes_description ::= **bytes** = integer

A protect encoding directive describes an encoding used for encrypted text in a decryption envelope.

If a protect encoding directive appears in an encryption envelope, other than in the source text, then the encryption tool shall use the encoding method to encode encrypted text in the corresponding decryption envelope. If an encryption envelope contains no protect encoding directive, a tool may choose an encoding method in an implementation-defined manner.

A protect encoding directive in a decryption envelope describes the encoding used in the immediately following key block, data block, or digest block.

The string literal following the **enctype** keyword identifies the encoding method (see 24.1.3.1).

The integer following the **line_length** keyword specifies the maximum number of characters, after encoding, that are permitted in each line of encoded text. For an encoding type other than "raw", a tool that encodes text shall insert end-of-line separators into the encoded text as follows:

a) If the standard or specification describing the encoding method specifies a fixed or maximum number of characters per line, the tool shall ensure that each line contains the fixed number, or no more than the maximum number, of characters. In this case, an encoding line length description has no effect.

b) If the standard or specification describing the encoding method does not specify a number of characters per line, then

— If a protect encoding directive in an encryption envelope contains an encoding line length description, the tool shall ensure that each line contains no more than the specified number of characters.

— Otherwise, the tool may choose a maximum line length in an implementation-defined manner and ensure that each line contains no more than that number of characters.

For the "raw" encoding type, an encoding line length description has no effect.

The integer following the **bytes** keyword specifies the number of octets, before encoding and insertion of end-of-line separators, in the unencoded text.

A directive in an encryption envelope may contain an encoding bytes description, but such an encoding bytes description has no effect. A protect encoding directive in a decryption envelope shall contain an encoding bytes description. Moreover, the directive shall contain an encoding line length description if the standard or specification describing the encoding method does not specify a number of characters per line.

24.1.2.23 Protect viewport directive

protect_viewport_directive :: =
` **protect viewport** = (viewport_object_description , viewport_access_description)

viewport_object_description ::= **object** = string_literal

viewport_access_description ::= **access** = string_literal

A protect viewport directive describes a declared object or objects in a VHDL description for which certain operations are to be permitted by a tool.

The value of the string literal following the **object** keyword identifies the declared object or objects. The value shall be in the form of the `DefName` property (see 19.4.6) of a VHPI object representing a declared object, but with the leading commercial at character, library name and period omitted. The declared objects identified are those declared objects for which both of the following hold:

— The declared object is declared in the source text of the protection envelope containing the protect viewport directive.

— The `DefName` property of the VHPI object representing the declared object, with the leading commercial at character, library name and period omitted, is the same as the value of the string literal, not counting differences in the use of corresponding uppercase and lowercase letters in basic identifiers.

If more than one declared object is identified, the operations are to be permitted for all of those declared objects.

The string literal following the **access** keyword specifies which operations are permitted for the identified declared object or objects, as follows:

— "R": *Read-only access*—The object may be read by any part of the VHDL description, by a VHPI program, or by a tool. It is an error if the object is updated other than by part of the VHDL description contained within the protection envelope containing the protect viewport directive.

— "W": *Write-only access*—The object may be updated by any part of the VHDL description, by a VHPI program, or by a tool. It is an error if the object is read other than by part of the VHDL description contained within the protection envelope containing the protect viewport directive.

— "RW": *Read-write access*—The object may be read and updated.

It is an error if any other string literal, other than one that differs only in the use of corresponding uppercase and lowercase letters, appears in a viewport access description.

If a protect viewport directive appears in an encryption envelope, other than in the source text, then the directive shall appear unchanged in the corresponding decryption envelope.

If a decryption tool is presented with more than one protect viewport directive for a given declared object, it is an error if the viewport access descriptions do not all specify the same access.

NOTE 1—The object description may identify more than one object if there are overloaded subprograms, each of which declares an object of a given name. The values of the `DefName` properties of those objects may be identical.

NOTE 2—A protect viewport directive can be placed in the source text of an encryption envelope to avoid disclosing information about the viewport to the tool user and to avoid modification of the directive. The directive is interpreted by the decryption tool after decrypting the decryption envelope. A duplicate viewport directive may also be placed outside the source text in the encryption envelope to serve as documentation for the tool user. The requirement that the two directives specify the same access can be used to detect alteration of the non-encrypted directive.

24.1.2.24 Protect license directives

protect_decrypt_license_directive ::= ` **protect decrypt_license** = license_description

protect_runtime_license_directive ::= ` **protect runtime_license** = license_description

license_description ::=
    (**library** = string_literal , **entry** = string_literal , **feature** = string_literal ,
      [**exit** = string_literal ,] **match** = integer)

A protect license directive provides information to be used by a tool to acquire one or more licenses. A protect decrypt license directive describes acquisition information for a license that allows a decryption tool to proceed with decryption of the enclosing VHDL description. A protect runtime license directive describes acquisition information for a license that allows a decryption tool to proceed with execution of the enclosing VHDL description.

The string literal following the **library** keyword identifies an object library. The mapping between the string value and a host physical object library is not defined by this standard. If the host system cannot locate the physical object library identified by the string value, acquisition of the license fails.

The string literal following the **entry** keyword identifies an entry point in the object library that can be called to acquire a license. It is an error if the host system cannot locate an entry point using the string value. The string literal following the **exit** keyword, if present, identifies an entry point in the object library that can be called to release a license. It is an error if the exit keyword and a string value are specified and the host system cannot locate an entry point using the string value.

A tool acquires a license described by a license description by calling the license acquisition entry point, passing as a parameter the value of the string literal that follows the **feature** keyword. The acquisition entry point shall return an integer value. The decryption tool shall compare the returned integer value with the value of the integer following the **match** keyword. If the values are equal, the tool is granted the license; otherwise, the tool is denied the license and may use the return value of the acquisition entry point in an error message or may pass the return value as a status value to the host system environment.

If a protect decrypt license directive or protect runtime license directive appears in an encryption envelope, other than in the source text, the directive shall appear unchanged in the corresponding decryption envelope.

If a protect decrypt license directive or protect runtime license directive appears in a decryption envelope or within a decrypted portion of a VHDL description, a decryption tool shall acquire the license described by

the directive. For a protect decrypt license directive, if the tool is granted the license, it may proceed with further analysis of the VHDL description and may decrypt any enclosing decryption envelope and subsequent decryption envelopes in the VHDL description. Upon completion of decryption, the tool shall call the license release entry point. If the tool fails to acquire the license or is denied the license, the tool shall not proceed with any further analysis or decryption of the VHDL description. For a protect runtime license directive, if the tool is granted the license, it may proceed to execute the VHDL description. Upon termination of execution, the tool shall call the license release entry point. If the tool fails to acquire the license or is denied the license, the tool shall not execute the VHDL description.

NOTE—A protect decrypt license directive may appear as part of the source text in an encryption envelope. In that case, it is encrypted as part of the source text. If a decryption tool successfully decrypts the text, it shall still acquire the decryption license. If acquisition fails or is denied, the tool was not supposed to have decrypted the source text and shall not proceed with further analysis.

24.1.2.25 Protect comment directive

protect_comment_directive :: = ` **protect comment** = string_literal

A protect comment directive provides information for the enlightenment of the human reader. If a protect comment directive appears in an encryption envelope, whether preceding or in the source text, then the directive shall appear unchanged in the corresponding decryption envelope. A protect comment directive appearing in the source text in an encryption envelope shall not be encrypted as part of the source text. If a protect comment directive appears in a decryption envelope, it has no effect on decryption of the decryption envelope.

24.1.3 Encoding, encryption, and digest methods

24.1.3.1 Encoding methods

This standard defines the following strings in encoding type descriptions and the corresponding encoding methods:

Encoding type string	Required/optional	Encoding methods
"uuencode"	Required	IEEE Std 1003.1™-2004 [B9] (uuencode Historical Algorithm)
"base64"	Required	IETF RFC 2045 [B19] {also IEEE Std 1003.1-2004 [B9] (uuencode-m)}
"quoted-printable"	Optional	IETF RFC 2045 [B19]
"raw"	Optional	Identity transformation; no encoding is performed, and the data may contain non-printing characters.

The encoding methods identified by required encoding type strings shall be implemented by a tool. A tool may implement an encoding method identified by an optional encoding type string, but if it does implement such a method, it shall use the corresponding encoding type string to identify that method.

A tool may implement further encoding methods and use other encoding type strings to identify those methods. Any further encoding method implemented by a tool should produce only printing graphic characters in the encoded text. Moreover, the tool, given the number of octets in the unencoded text, should be able to determine the exact number of characters of encoded text required to be decoded to yield the unencoded text. The effect of use of an encoding method that does not meet these conditions is not specified by this standard.

It is an error if a protect directive identifies an encoding method that is not implemented by a tool processing the protect directive.

NOTE—The text produced by "raw" encoding may contain characters that signify the end of a line in some implementations. Transmission of a VHDL description containing such characters between host systems may involve translation of the characters, thus changing the content, length, or both, of the encrypted text. A change in the length of the text may cause an error when the text is read by a decryption tool.

24.1.3.2 Encryption methods

This standard defines the following strings in encryption method descriptions and the corresponding ciphers:

Encryption method string	Required/ optional	Cipher	Cipher type
"des-cbc"	Required	DES in CBC mode (FIPS PUB 46-3 [B4], FIPS PUB 81 [B5]).	Symmetric
"3des-cbc"	Optional	Triple DES in CBC mode (ANSI X9.52 [B2], FIPS PUB 46-3 [B4]).	Symmetric
"aes128-cbc"	Optional	AES in CBC mode with 128-bit key (FIPS PUB 197 [B7]).	Symmetric
"aes192-cbc"	Optional	AES in CBC mode with 192-bit key (FIPS PUB 197 [B7]).	Symmetric
"aes256-cbc"	Optional	AES in CBC mode with 256-bit key (FIPS PUB 197 [B7]).	Symmetric
"blowfish-cbc"	Optional	Blowfish in CBC mode (Schneier [B24]).	Symmetric
"twofish128-cbc"	Optional	Twofish in CBC mode with 128-bit key (Schneier et al. [B25]).	Symmetric
"twofish192-cbc"	Optional	Twofish in CBC mode with 192-bit key (Schneier et al. [B25]).	Symmetric
"twofish256-cbc"	Optional	Twofish in CBC mode with 256-bit key (Schneier et al. [B25]).	Symmetric
"serpent128-cbc"	Optional	Serpent in CBC mode with 128-bit key (Anderson [B1]).	Symmetric
"serpent192-cbc"	Optional	Serpent in CBC mode with 192-bit key (Anderson [B1]).	Symmetric
"serpent256-cbc"	Optional	Serpent in CBC mode with 256-bit key (Anderson [B1]).	Symmetric
"cast128-cbc"	Optional	CAST-128 in CBC mode (IETF RFC 2144 [B20]).	Symmetric
"rsa"	Optional	RSA (IETF RFC 2437 [B21]).	Asymmetric
"elgamal"	Optional	ElGamal [B3].	Asymmetric
"pgp-rsa"	Optional	OpenPGP RSA key (IETF RFC 2440 [B22]).	Asymmetric

The ciphers identified by required encryption method strings shall be implemented by a tool. A tool may implement a cipher identified by an optional encryption method string, but if it does implement such a cipher, it shall use the corresponding encryption method string to identify that cipher. A tool may implement further ciphers and use other encryption method strings to identify those ciphers.

If a symmetric cipher is used in *cipher-block chaining* (CBC) mode, requiring an *initialization vector*, the encryption tool shall generate the initialization vector and include it as the first block of the encrypted information. It is recommended that the initialization vector be randomly generated for each use of the cipher.

It is an error if a protect directive identifies a cipher that is not implemented by a tool processing the protect directive.

NOTE—Use of a symmetric cipher to encrypt a session key in a digital envelope is not a common use case. Nonetheless, should such a cipher be used for that purpose, an initialization vector must be generated and included as the first block of encrypted information in the decrypt key block.

24.1.3.3 Digest methods

This standard defines the following strings in digest method descriptions and the corresponding hash functions:

Digest method string	Required/optional	Hash function
"sha1"	Required	Secure Hash Algorithm 1 (SHA-1) (FIPS PUB 180-3 [B6]).
"md5"	Required	Message Digest Algorithm 5 (IETF RFC 1321 [B18]).
"md2"	Optional	Message Digest Algorithm 2 (IETF RFC 1319 [B17]).
"ripemd-160"	Optional	RIPEMD-160 (ISO/IEC 10118-3 [B23]).

The hash functions identified by required digest method strings shall be implemented by a tool. A tool may implement a hash function identified by an optional digest method string, but if it does implement such a hash function, it shall use the corresponding digest method string to identify that hash function. A tool may implement further hash functions and use other digest method strings to identify those hash functions.

It is an error if a protect directive identifies a hash function that is not implemented by a tool processing the protect directive.

24.1.4 Encryption envelopes

24.1.4.1 General

An encryption envelope contains a portion of a VHDL description to be encrypted, along with protection directives that specify how the text of that portion is to be encrypted. A tool that performs such encryption is called an *encryption tool*. An encryption tool processes a design file containing one or more encryption envelopes and produces a design file in which each encryption envelope is replaced by a corresponding decryption envelope, and other text is unchanged. The tool may store the resulting design file in an implementation-defined manner.

```
encryption_envelope ::=
    { encrypt_specification }
    protect_begin_directive
        source_text
    protect_end_directive

encrypt_specification ::=
      encrypt_author_specification
    | encrypt_key_specification
```

```
    | encrypt_data_specification
    | encrypt_digest_specification
    | encrypt_license_specification
    | protect_encoding_directive
    | protect_viewport_directive

encrypt_author_specification ::=
      protect_author_directive [ protect_author_info_directive ]
    | protect_author_info_directive [ protect_author_directive ]

encrypt_license_specification ::=
      protect_decrypt_license_directive [ protect_runtime_license_directive ]
    | protect_runtime_license_directive [ protect_decrypt_license_directive ]
```

The protect directives in an encryption envelope may be combined into protect directives with multiple keyword expressions (see 24.1.1), provided the equivalent sequence of protect directives each containing one keyword expression conforms to the rules for forming an encryption envelope.

An encryption envelope may contain protect comment directives within or between any specifications or directives, or within the source text. Such protect comment directives do not form part of a specification or part of the source text, but are included unchanged in the corresponding decryption envelope.

The source text in an encryption envelope is a sequence of lexical elements and separators (see 15.3). The encryption tool performs no analysis on the lexical elements, other than determining that the text is properly composed of lexical elements and separators, identifying protect comment directives, and locating the first protect end directive, which indicates the end of the source text.

It is an error if an encryption envelope contains more than one of each of an encrypt author specification, an encrypt license specification, an encrypt data specification, an encrypt digest specification, or a protect encoding directive.

NOTE 1—Encryption envelopes cannot be nested. All characters between a protect begin directive and the first subsequent protect end directive, including any characters that would otherwise form an encryption envelope, but excluding any protect comment directives, are treated as text to be encrypted.

NOTE 2—The text to be encrypted may include a decryption envelope. Since all characters between the protect begin and protect end directives of an encryption envelope, other than characters in a protect comment directive, form text to be encrypted, any protect directives in the decryption envelope are not interpreted when processing the encryption envelope. The result of encrypting the text is a decryption envelope containing a nested encrypted decryption envelope.

NOTE 3—If an implementation uses one or more characters to signify the end of a line, any such characters occurring between protect begin and protect end directives are included as part of the text to be encrypted. If the corresponding decryption envelope is decrypted on another implementation that signifies the end of a line differently, the number of lines in the resulting text may be different. See 15.3.

24.1.4.2 Encrypt key specifications

```
encrypt_key_specification ::=
    { encrypt_key_directive }
    protect_key_block_directive

encrypt_key_directive
      protect_key_keyowner_directive
    | protect_key_keyname_directive
    | protect_key_method_directive
```


An encrypt key specification shall contain at most one of each of the encrypt key directives. If any encrypt key directive appears, then both a protect key keyowner directive and a protect key method directive shall appear.

If an encrypt key specification occurs in an encryption envelope, then the encryption tool shall form a digital envelope in the corresponding decryption envelope. The tool shall use a symmetric cipher to encrypt the source text and shall choose a session key in an implementation-defined manner for use with that cipher. The tool shall also include a decrypt key block in the corresponding decryption envelope that contains the encoded encrypted session key and the protect directives required to decode and decrypt the session key.

If no encrypt key directives appear in an encrypt key specification, then the encryption tool chooses a cipher and a key in an implementation-defined manner to encrypt the session key. Otherwise, the encryption tool shall use the cipher and key identified by the encrypt key directives to encrypt the session key. If the cipher is an asymmetric cipher, the public key of the identified key pair is used.

If more than one encrypt key specification occurs in an encryption envelope, the encryption tool shall choose only one session key to encrypt the source text. The tool shall include a decrypt key block, as described in this subclause, for each encrypt key specification appearing in the encryption envelope.

24.1.4.3 Encrypt data specifications

```
encrypt_data_specification ::=
    encrypt_data_directive
    { encrypt_data_directive }

encrypt_data_directive ::=
      protect_data_keyowner_directive
    | protect_data_keyname_directive
    | protect_data_method_directive
```

If an encryption envelope contains one or more encrypt key specifications, then any encrypt data specification in that envelope shall contain exactly one protect data method directive and no other encrypt data directive. The protect data method directive, if present, shall identify a symmetric cipher, and that cipher is used to encrypt the source text using the session key chosen by the tool.

If an encryption envelope contains no encrypt key specification, then any encrypt data specification in that envelope shall contain exactly one protect data keyowner directive and exactly one protect data method directive, and at most one protect data keyname directive. The protect data method directive may identify a symmetric cipher or an asymmetric cipher. The encryption tool shall use the cipher and key identified by the encrypt data directives to encrypt the source text. If the cipher is an asymmetric cipher, the public key of the identified key pair is used.

If an encryption envelope contains no encrypt data specification, then the encryption tool chooses a cipher in an implementation-defined manner to encrypt the source text. If the encryption envelope contains one or more encrypt key specifications, then the tool uses the chosen session key with the chosen cipher. Otherwise, the tool chooses a key in an implementation-defined manner for use with the chosen cipher.

The encryption tool shall include a decrypt data block in the corresponding decryption envelope containing the encoded encrypted source text and the protect directives required to decode and decrypt the source text.

24.1.4.4 Encrypt digest specifications

```
encrypt_digest_specification ::=
    { encrypt_digest_directive }
```

```
    protect_digest_block_directive

encrypt_digest_directive ::=
      protect_digest_keyowner_directive
    | protect_digest_keyname_directive
    | protect_digest_key_method_directive
    | protect_digest_method_directive
```

An encrypt digest specification shall contain at most one of each of the encrypt digest directives. If a protect digest keyowner directive appears, then a protect digest key method directive shall appear and a protect digest keyname directive may appear. It is an error if a protect digest keyname directive appears and there is no protect digest keyowner directive.

If an encrypt digest specification occurs in an encryption envelope, then the encryption tool shall form a digital signature in the corresponding decryption envelope. The tool shall use a hash function to compute a digest of the source text and shall encrypt the digest with an asymmetric cipher using a private key. The tool shall also include a decrypt digest block in the corresponding decryption envelope that contains the encoded encrypted digest and the protect directives required to decode and decrypt the digest.

If no protect digest method directive appears in an encrypt digest specification, then the encryption tool chooses a hash function in an implementation-defined manner to compute the digest. Otherwise, the encryption tool shall use the hash function identified by the protect digest method directive to compute the digest.

If no protect digest keyowner directive and protect digest key method directive appear in an encrypt digest specification, then the encryption tool chooses a cipher and a key in an implementation-defined manner to encrypt the digest. Otherwise, the encryption tool shall use the cipher and key identified by the directives to encrypt the digest.

24.1.5 Decryption envelopes

24.1.5.1 General

A decryption envelope contains an encrypted portion of a VHDL description, along with protection directives that specify how the text of that portion is to be decrypted. A tool that performs such decryption is called a *decryption tool*. A decryption tool processes a design file containing one or more decryption envelopes and produces a design file with each decryption envelope replaced by the decrypted text, and with other text unchanged. The resulting design file may be further analyzed and interpreted by the tool or may be stored in an implementation-defined manner, provided the decrypted text is not disclosed to the user of the tool.

```
decryption_envelope ::=
    protect_begin_protected_directive
    [ decrypt_author_specification ]
    [ decrypt_license_specification ]
    decrypt_encrypt_agent_specification
    { protect_viewport_directive }
    { decrypt_key_block }
    decrypt_data_block
    [ decrypt_digest_block ]
    protect_end_protected_directive

decrypt_author_specification ::=
      protect_author_directive [ protect_author_info_directive ]
```


| protect_author_info_directive [protect_author_directive]

decrypt_license_specification ::=
protect_decrypt_license_directive [protect_runtime_license_directive]
| protect_runtime_license_directive [protect_decrypt_license_directive]

decrypt_encrypt_agent_specification ::=
protect_encrypt_agent_directive [protect_encrypt_agent_info_directive]
| protect_encrypt_agent_info_directive protect_encrypt_agent_directive

The protect directives in a decryption envelope may be combined into protect directives with multiple keyword expressions (see 24.1.1), provided the equivalent sequence of protect directives each containing one keyword expression conforms to the rules for forming a decryption envelope.

A decryption envelope may contain protect comment directives, including those that appear in any part of the corresponding encryption envelope, within or between any specifications or directives. Such protect comment directives do not form part of a specification.

The encoded text in a decrypt key block, a decrypt data block, or a decrypt digest block is a sequence of characters (see 24.1.3.1).

If a decrypted portion of a design file contains further decryption envelopes, the decryption tool shall further process those decryption envelopes as described in this subclause (24.1.5).

24.1.5.2 Decrypt key blocks

decrypt_key_block ::=
protect_key_keyowner_directive
[protect_key_keyname_directive]
protect_key_method_directive
protect_encoding_directive
protect_key_block_directive
encoded_text

A decrypt key block in a decryption envelope contains an encoded encrypted session key for a digital envelope. A decryption tool shall determine in an implementation-defined manner whether it has access to the key identified by the protect key keyowner directive and the protect key keyname directive (if present). If the cipher identified by the protect key method directive is an asymmetric cipher, the tool determines whether is has access to the private key of the identified key pair.

If more than one decrypt key block appears in a decryption envelope, the decryption tool shall determine whether it has access to the key identified by any of the decrypt key blocks. If may choose any of the decrypt key blocks for which it has access to the identified key to obtain the session key.

For a given protect key block, if the tool has access to the identified key, it uses the encoding identified by the protect encoding directive to decode the encoded text to obtain the encrypted session key. The tool then uses the cipher identified by the protect key method directive with the identified key to decrypt the session key.

24.1.5.3 Decrypt data blocks

decrypt_data_block ::=
[protect_data_keyowner_directive
[protect_data_keyname_directive]]

protect_data_method_directive
protect_encoding_directive
protect_data_block_directive
encoded_text

A decrypt data block in a decryption envelope contains an encoded encrypted portion of a VHDL description. If the decryption envelope contains one or more decrypt key blocks, the key used for decryption is a session key, and a decryption tool shall obtain the session key as described in 24.1.5.2. It is an error if the decryption envelope contains one or more decrypt key blocks and the decrypt data block contains a protect data keyowner directive.

If the decryption envelope contains no decrypt key blocks, then the decrypt data block shall contain a protect data keyowner directive and may contain a protect data keyname directive. The key identified by the protect data keyowner directive and the protect data keyname directive (if present) is the key used to decrypt the encrypted portion of the VHDL description. If the cipher identified by the protect data method directive is an asymmetric cipher, the private key of the identified key pair is used. It is an error if the decryption tool does not have access to the identified key.

The decryption tool uses the encoding identified by the protect encoding directive to decode the encoded text to obtain the encrypted portion of the VHDL description. The tool then uses the cipher identified by the protect data method directive with the session key or identified key, as appropriate, to decrypt the portion of the VHDL description.

24.1.5.4 Decrypt digest blocks

decrypt_digest_block ::=
protect_digest_keyowner_directive
[protect_digest_keyname_directive]
protect_digest_key_method_directive
protect_digest_method_directive
protect_encoding_directive
protect_digest_block_directive
encoded_text

A decrypt digest block in a decryption envelope contains an encoded digital signature of a portion of a VHDL description. A decryption tool shall determine in an implementation-defined manner whether it has access to the public key of the key pair identified by the protect digest keyowner directive and the protect digest keyname directive (if present). It is an error if the tool does not have access to the key. Otherwise, the tool shall verify the digital signature as follows:

a) The tool shall compute a digest of the decrypted portion of the VHDL description (see 24.1.5.3) using the hash function identified by the decrypt digest method directive.
b) The tool shall use the encoding identified by the protect encoding directive to decode the encoded text to obtain the encrypted signature digest, and then use the cipher identified by the protect digest key method directive with the identified public key to decrypt the signature digest.
c) It is an error if the computed digest differs from the signature digest.

24.1.6 Protection requirements for decryption tools

Since the purpose of encrypting portions of a VHDL description is to prevent disclosure of those portions to a user, a decryption tool, after processing a decryption envelope, shall conform to the following restriction, unless otherwise permitted in an implementation-defined manner by the effect of acquiring a license (see 24.1.2.24):

— A decryption tool shall not display or store in any form accessible to the user or other tools any parts of decrypted portions of a VHDL description, decrypted keys, or decrypted digests.

— If a decryption tool transforms a decrypted portion of a VHDL description (for example, by synthesizing a circuit and describing it in VHDL or any other representation), then these requirements shall apply to the transformed portion also.

— If a decryption tool provides a means for the tool user to gain access to a representation of a VHDL description or an elaboration of a VHDL description (for example, by means of a user interface or an applications programming interface such as VHPI), then the tool shall not provide access to any representation of a decrypted portion of a VHDL description other than a representation of an object specified by a protect viewport directive.

— Any message (for example, an error message) generated by a decryption tool shall not include information that discloses content of a decrypted portion of a VHDL description, a decrypted key, or a decrypted digest. For example, a message shall not include a name or hierarchical path name identifying part of a decrypted portion of a VHDL description.

— If a decryption tool executes an assertion statement (see 10.3) that causes an assertion violation, or executes a report statement (see 10.4), the message shall not include the name of the design unit containing the statement, the rules of 10.3 and 10.4 notwithstanding.

— The value of any 'INSTANCE_NAME or 'PATH_NAME predefined attribute (see 16.2) formed by the decryption tool shall not include any element that is a name or label defined in a decrypted portion of a VHDL description, the rules of 16.2 notwithstanding.

— If a decrypted portion of a VHDL description includes an instantiation of a declaration that is declared in a portion of the VHDL description that is not encrypted, a decryption tool may provide access to a representation of the design subhierarchy whose root is the instance, provided the means of providing access does not contradict other requirements of this subclause. For example, a VHPI tool may return a handle to a VHPI object representing such an instance and allow navigation of associations from that reference object, provided the target objects represent parts of the design subhierarchy.

Annex A

(informative)

Description of accompanying files

A.1 General

This annex describes the machine-readable elements that accompany this standard document. They are included in an archive file available for download from the IEEE Web site.[15]

A.2 Package bodies

A.2.1 General

VHDL source files for the packages residing in library IEEE described in this standard are in the `ieee` directory of the archive file. The bodies of the standard mathematical packages are not normative. They are provided as a guideline to implementers, to suggest ways in which implementers may implement the packages. The bodies of the standard multivalue package and the standard synthesis packages, on the other hand, are normative specifications of the semantics of the packages.

A.2.2 Standard mathematical packages

The VHDL source files for the standard mathematical packages are:

- `math_real.vhdl`: package declaration for MATH_REAL
- `math_real-body.vhdl`: package body for MATH_REAL
- `math_complex.vhdl`: package declaration for MATH_COMPLEX
- `math_complex-body.vhdl`: package body for MATH_COMPLEX

A.2.3 Standard multivalue logic package

The VHDL source files for the standard multivalue logic package are:

- `std_logic_1164.vhdl`: package declaration for STD_LOGIC_1164
- `std_logic_1164-body.vhdl`: package body for STD_LOGIC_1164
- `std_logic_textio.vhdl`: package declaration for STD_LOGIC_TEXTIO

A.2.4 Standard synthesis packages

The VHDL source files for the standard synthesis packages are:

- `numeric_bit.vhdl`: package declaration for NUMERIC_BIT
- `numeric_bit-body.vhdl`: package body for NUMERIC_BIT
- `numeric_std.vhdl`: package declaration for NUMERIC_STD
- `numeric_std-body.vhdl`: package body for NUMERIC_STD

[15]The archive file is available at http://standards.ieee.org/downloads/1076/1076-2008/.

— `numeric_bit_unsigned.vhdl`: package declaration for NUMERIC_BIT_UNSIGNED
— `numeric_bit_unsigned-body.vhdl`: package body for NUMERIC_BIT_UNSIGNED
— `numeric_std_unsigned.vhdl`: package declaration for NUMERIC_STD_UNSIGNED
— `numeric_std_unsigned-body.vhdl`: package body for NUMERIC_STD_UNSIGNED

A.2.5 Fixed-point and floating-point packages

The VHDL source files for the fixed-point and floating-point packages are:

— `fixed_float_types.vhdl`: package declaration for FIXED_FLOAT_TYPES
— `fixed_generic_pkg.vhdl`: package declaration for FIXED_GENERIC_PKG
— `fixed_generic_pkg-body.vhdl`: package body for FIXED_GENERIC_PKG
— `fixed_pkg.vhdl`: package instantiation declaration for FIXED_PKG
— `float_generic_pkg.vhdl`: package declaration for FLOAT_GENERIC_PKG
— `float_generic_pkg-body.vhdl`: package body for FLOAT_GENERIC_PKG
— `float_pkg.vhdl`: package instantiation declaration for FLOAT_PKG

A.3 Testbench files

A.3.1 General

VHDL source files for testbenches for the standard packages are in the `testbench` directory of the archive file. These testbenches are not normative, and do not exhaustively test the packages. They are provided as an aid to implementers developing alternative implementations of the package bodies.

A.3.2 Testbenches for the standard mathematical packages

The VHDL source files for the testbenches for the standard mathematical packages are:

— `real_tests.vhd`: testbench for MATH_REAL
— `complex_tests.vhd`: testbench for MATH_COMPLEX

A.4 VHPI files

A.4.1 Machine-readable information model

The VHPI information model was developed using a specialized software tool[16] and is represented in XMI 2.1 format. The XMI file, `vhpi_uml.xml`, is provided in the `vhpi_uml` directory of the archive file.

The archive file also contains a browsable HTML model report of the information model, generated using the specialized software tool. The main index file of the report is `vhpi_uml.html`, located in the `html` directory within the `vhpi_uml` directory.

[16]The following information is given for the convenience of users of this standard and does not consittute an endorsement by the IEEE of this product. Equivalent products may be used if they can be shown to lead to the same results. The specialized software tool is MagicDraw® provided by No Magic, Inc.® MagicDraw is available for free download from www.magicdraw.com. Other UML tools may be able to read the XMI representation of the information model, though they might not be able to interpret the representation of the diagrams.

A.4.2 VHPI header file

The VHPI header file, `vhpi_user.h`, is provided in the `code` directory of the archive file.

A.4.3 VHPI definitions file

A.4.3.1 General

The VHPI definitions file, `vhpi_def.c`, is provided in the `code` directory of the archive file. This C source file contains a definition for the `vhpi_is_printable` function (see 23.3) and other definitions described in this subclause (A.4.3). The file is informative and is provided as a guide to implementers of VHPI tools.

A.4.3.2 VHPICharCodes

An array of strings of graphic characters corresponding to character codes.

Synopsis:

```
static const char* VHPICharCodes[256] = {
  "NUL",  "SOH",  "STX",  "ETX",  "EOT",  "ENQ",  "ACK",  "BEL",
  "BS",   "HT",   "LF",   "VT",   "FF",   "CR",   "SO" ,  "SI",
  "DLE",  "DC1",  "DC2",  "DC3",  "DC4",  "NAK",  "SYN",  "ETB",
  "CAN",  "EM",   "SUB",  "ESC",  "FSP",  "GSP",  "RSP",  "USP",
  " ",    "!",    "\"",   "#",    "$",    "%",    "&",    "'",
  "(",    ")",    "*",    "+",    ",",    "-",    ".",    "/",
  "0",    "1",    "2",    "3",    "4",    "5",    "6",    "7",
  "8",    "9",    ":",    ";",    "<",    "=",    ">",    "?",
  "@",    "A",    "B",    "C",    "D",    "E",    "F",    "G",
  "H",    "I",    "J",    "K",    "L",    "M",    "N",    "O",
  "P",    "Q",    "R",    "S",    "T",    "U",    "V",    "W",
  "X",    "Y",    "Z",    "[",    "\\",   "]",    "^",    "_",
  "`",    "a",    "b",    "c",    "d",    "e",    "f",    "g",
  "h",    "i",    "j",    "k",    "l",    "m",    "n",    "o",
  "p",    "q",    "r",    "s",    "t",    "u",    "v",    "w",
  "x",    "y",    "z",    "{",    "|",    "}",    "~",    "DEL",
  "C128", "C129", "C130", "C131", "C132", "C133", "C134", "C135",
  "C136", "C137", "C138", "C139", "C140", "C141", "C142", "C143",
  "C144", "C145", "C146", "C147", "C148", "C149", "C150", "C151",
  "C152", "C153", "C154", "C155", "C156", "C157", "C158", "C159",
  " ",    "¡",    "¢",    "£",    "¤",    "¥",    "¦",    "§",
  "¨",    "©",    "ª",    "«",    "¬",    "-",    "®",    "¯",
  "°",    "±",    "²",    "³",    "´",    "µ",    "¶",    "·",
  "¸",    "¹",    "º",    "»",    "¼",    "½",    "¾",    "¿",
  "À",    "Á",    "Â",    "Ã",    "Ä",    "Å",    "Æ",    "Ç",
  "È",    "É",    "Ê",    "Ë",    "Ì",    "Í",    "Î",    "Ï",
  "Ð",    "Ñ",    "Ò",    "Ó",    "Ô",    "Õ",    "Ö",    "×",
  "Ø",    "Ù",    "Ú",    "Û",    "Ü",    "Ý",    "Þ",    "ß",
  "à",    "á",    "â",    "ã",    "ä",    "å",    "æ",    "ç",
  "è",    "é",    "ê",    "ë",    "ì",    "í",    "î",    "ï",
  "ð",    "ñ",    "ò",    "ó",    "ô",    "õ",    "ö",    "÷",
  "ø",    "ù",    "ú",    "û",    "ü",    "ý",    "þ",    "ÿ" };
```

Description:

Each element of the array is a null-terminated string whose value is a printable representation of the character code that is the index of the element. For character codes representing graphic characters (see 15.2), the string contains just the graphic character. For other character codes, the string contains a representation in uppercase letters of the enumeration literal of type STD.STANDARD.CHARACTER whose position number is the character code.

A.4.3.3 VHPI_GET_PRINTABLE_STRINGCODE

Gets a string of graphic characters corresponding to a character code.

Synopsis:

```
#define VHPI_GET_PRINTABLE_STRINGCODE( ch ) VHPICharCodes[unsigned char ch]
```

Description:

The macro takes as its argument a character code in the range 0 to 255 and substitutes an expression that uses the character code to index an element of the `VHPICharCodes` array (see A.4.3). The type of the expression is a pointer to a null-terminated string.

A.4.4 VHPI sensitivity-set functions file

The VHPI sensitivity-set function file, `vhpi_sens.c`, is provided in the `code` directory of the archive file. This C source file contains definitions of functions that implement the sensitivity-set macros (see B.2). The file also contains a main program that can be used to perform regression testing of the functions. The file is informative and is provided as a guide to implementers of VHPI tools.

Annex B

(normative)

VHPI header file

B.1 General

The VHPI header file, `vhpi_user.h`, shall be included by a VHPI tool. A tool provider should provide the header file with the tool.

Several definitions in the VHPI header file are marked as deprecated. They are included for compatibility with earlier versions of the VHPI than that defined by this standard. VHPI programs that conform to this standard should not use definitions so marked. The function `vhpi_get_foreign_info`, which is marked as deprecated, is defined to be the same as the `vhpi_get_foreignf_info` function. The deprecated function will be removed in a future revision of this standard.

The content of the vhpi_user.h file is as follows:

```
/* ------------------------------------------------------------------
/*
/* Copyright © 2008 by IEEE. All rights reserved.
/*
/* This source file is an essential part of IEEE Std 1076-2008,
/* IEEE Standard VHDL Language Reference Manual. This source file may
/* not be copied, sold, or included with software that is sold without
/* written permission from the IEEE Standards Department. This source
/* file may be copied for individual use between licensed users. This
/* source file is provided on an AS IS basis. The IEEE disclaims ANY
/* WARRANTY EXPRESS OR IMPLIED INCLUDING ANY WARRANTY OF
/* MERCHANTABILITY AND FITNESS FOR USE FOR A PARTICULAR PURPOSE. The
/* user of the source file shall indemnify and hold IEEE harmless from
/* any damages or liability arising out of the use thereof.
/*
/*   Title     :  vhpi_user.h
/*             :
/*   Developers:  IEEE P1076 Working Group, VHPI Task Force
/*             :
/*   Purpose   :  This header file describes the procedural interface
/*             :  to access VHDL compiled, instantiated and run-time
/*            :  data. It is derived from the UML model. For conformance
/*             :  with the VHPI standard, a VHPI application or program
/*             :  shall reference this header file.
/*             :
/*   Note      :  The contents of this file may be modified in an
/*             :  implementation to provide implementation-defined
/*             :  functionality, as described in B.3.
/*             :
/* ------------------------------------------------------------------
/* modification history :
```

```
/* ------------------------------------------------------------------
/* $Revision: 1387 $
/* $Date: 2009-01-10 17:45:30 +1030 (Sat, 10 Jan 2009) $
/* ------------------------------------------------------------------
 */


#ifndef VHPI_USER_H
#define VHPI_USER_H
#include <stddef.h>
#include <stdarg.h>
/* Ensure that size-critical types are defined on all OS platforms. */
#if defined (_MSC_VER)
typedef unsigned __int64 uint64_t;
typedef unsigned __int32 uint32_t;
typedef unsigned __int8 uint8_t;
typedef signed __int64 int64_t;
typedef signed __int32 int32_t;
typedef signed __int8 int8_t;
#elif defined(__MINGW32__)
#include <stdint.h>
#elif defined(__linux)
#include <inttypes.h>
#else
#include <sys/types.h>
#endif

#ifdef  __cplusplus
extern "C" {
#endif

/*--------------------------------------------------------------------*/
/*--------------------------- Portability Help -----------------------*/
/*--------------------------------------------------------------------*/
/* Use to export a symbol */
#if defined (_MSC_VER)
#ifndef PLI_DLLISPEC
#define PLI_DLLISPEC __declspec(dllimport)
#define VHPI_USER_DEFINED_DLLISPEC 1
#endif
#else
#ifndef PLI_DLLISPEC
#define PLI_DLLISPEC
#endif
#endif

/* Use to import a symbol */
#if defined (_MSC_VER)
#ifndef PLI_DLLESPEC
#define PLI_DLLESPEC __declspec(dllexport)
#define VHPI_USER_DEFINED_DLLESPEC 1
#endif
#else
#ifndef PLI_DLLESPEC
```

```
#define PLI_DLLESPEC
#endif
#endif

/* Use to mark a function as external */
#ifndef PLI_EXTERN
#define PLI_EXTERN
#endif

/* Use to mark a variable as external */
#ifndef PLI_VEXTERN
#define PLI_VEXTERN extern
#endif

#ifndef PLI_PROTOTYPES
#define PLI_PROTOTYPES
/* object is defined imported by the application */
#define XXTERN PLI_EXTERN PLI_DLLISPEC
/* object is exported by the application */
#define EETERN PLI_EXTERN PLI_DLLESPEC
#endif

/* basic typedefs */
#ifndef VHPI_TYPES
#define VHPI_TYPES
typedef uint32_t *vhpiHandleT;
typedef uint32_t vhpiEnumT;
typedef uint8_t vhpiSmallEnumT;
typedef int32_t vhpiIntT;
typedef int64_t vhpiLongIntT;
typedef unsigned char vhpiCharT;
typedef double vhpiRealT;
typedef int32_t vhpiSmallPhysT;
typedef struct vhpiPhysS
{
  int32_t high;
  uint32_t low;
} vhpiPhysT;

/********************** time structure ***************************/
typedef struct vhpiTimeS
{
  int32_t high;
  uint32_t low;
} vhpiTimeT;

/********************** value structure **************************/

/* value formats */
typedef enum {
  vhpiBinStrVal        = 1, /* do not move */
  vhpiOctStrVal        = 2, /* do not move */
  vhpiDecStrVal        = 3, /* do not move */
  vhpiHexStrVal        = 4, /* do not move */
```

```
  vhpiEnumVal          = 5,
  vhpiIntVal           = 6,
  vhpiLogicVal         = 7,
  vhpiRealVal          = 8,
  vhpiStrVal           = 9,
  vhpiCharVal          = 10,
  vhpiTimeVal          = 11,
  vhpiPhysVal          = 12,
  vhpiObjTypeVal       = 13,
  vhpiPtrVal           = 14,
  vhpiEnumVecVal       = 15,
  vhpiIntVecVal        = 16,
  vhpiLogicVecVal      = 17,
  vhpiRealVecVal       = 18,
  vhpiTimeVecVal       = 19,
  vhpiPhysVecVal       = 20,
  vhpiPtrVecVal        = 21,
  vhpiRawDataVal       = 22,
  vhpiSmallEnumVal     = 23,
  vhpiSmallEnumVecVal  = 24,
  vhpiLongIntVal       = 25,
  vhpiLongIntVecVal    = 26,
  vhpiSmallPhysVal     = 27,
  vhpiSmallPhysVecVal  = 28

#ifdef VHPIEXTEND_VAL_FORMATS
  VHPIEXTEND_VAL_FORMATS
#endif

} vhpiFormatT;

/* value structure */
typedef struct vhpiValueS
{
  vhpiFormatT format;  /* vhpi[Char,[Bin,Oct,Dec,Hex]Str,
                              [Small]Enum,Logic,Int,Real,
                              [Small]Phys,Time,Ptr,
                              [Small]EnumVec,LogicVec,IntVect,RealVec,
                              [Small]PhysVec,TimeVec,
                             PtrVec,ObjType,RawData]Val */
  size_t bufSize;  /* the size in bytes of the value buffer;
                      this is set by the user */
  int32_t numElems;
  /* different meanings depending on the format:
     vhpiStrVal, vhpi{Bin...}StrVal: size of string
     array type values: number of array elements
     scalar type values: undefined
  */

  vhpiPhysT unit;
  union
    {
      vhpiEnumT enumv, *enumvs;
      vhpiSmallEnumT smallenumv, *smallenumvs;
```

```
      vhpiIntT  intg, *intgs;
      vhpiLongIntT  longintg, *longintgs;
      vhpiRealT real, *reals;
      vhpiSmallPhysT smallphys, *smallphyss;
      vhpiPhysT phys, *physs;
      vhpiTimeT time, *times;
      vhpiCharT ch, *str;
      void *ptr, **ptrs;
    } value;
} vhpiValueT;

#endif

/* Following are the constant definitions. They are divided into
   three major areas:

 1) object types

 2) access methods

 3) properties

*/
#define vhpiUndefined -1

/*************** OBJECT KINDS ******************/
typedef enum {
        vhpiAccessTypeDeclK = 1001,
        vhpiAggregateK = 1002,
        vhpiAliasDeclK = 1003,
        vhpiAllK = 1004,
        vhpiAllocatorK = 1005,
        vhpiAnyCollectionK = 1006,
        vhpiArchBodyK = 1007,
        vhpiArgvK = 1008,
        vhpiArrayTypeDeclK = 1009,
        DEPRECATED_vhpiAssertStmtK = 1010,
        vhpiAssocElemK = 1011,
        vhpiAttrDeclK = 1012,
        vhpiAttrSpecK = 1013,
        DEPRECATED_vhpiBinaryExprK = 1014,
        vhpiBitStringLiteralK = 1015,
        vhpiBlockConfigK = 1016,
        vhpiBlockStmtK = 1017,
        vhpiBranchK = 1018,
        vhpiCallbackK = 1019,
        vhpiCaseStmtK = 1020,
        vhpiCharLiteralK = 1021,
        vhpiCompConfigK = 1022,
        vhpiCompDeclK = 1023,
        vhpiCompInstStmtK = 1024,
        vhpiCondSigAssignStmtK = 1025,
        vhpiCondWaveformK = 1026,
        vhpiConfigDeclK = 1027,
```

```
    vhpiConstDeclK = 1028,
    vhpiConstParamDeclK = 1029,
    vhpiConvFuncK = 1030,
    vhpiDerefObjK = 1031,
    vhpiDisconnectSpecK = 1032,
    vhpiDriverK = 1033,
    vhpiDriverCollectionK = 1034,
    vhpiElemAssocK = 1035,
    vhpiElemDeclK = 1036,
    vhpiEntityClassEntryK = 1037,
    vhpiEntityDeclK = 1038,
    vhpiEnumLiteralK = 1039,
    vhpiEnumRangeK = 1040,
    vhpiEnumTypeDeclK = 1041,
    vhpiExitStmtK = 1042,
    vhpiFileDeclK = 1043,
    vhpiFileParamDeclK = 1044,
    vhpiFileTypeDeclK = 1045,
    vhpiFloatRangeK = 1046,
    vhpiFloatTypeDeclK = 1047,
    vhpiForGenerateK = 1048,
    vhpiForLoopK = 1049,
    vhpiForeignfK = 1050,
    vhpiFuncCallK = 1051,
    vhpiFuncDeclK = 1052,
    vhpiGenericDeclK = 1053,
    vhpiGroupDeclK = 1054,
    vhpiGroupTempDeclK = 1055,
    vhpiIfGenerateK = 1056,
    vhpiIfStmtK = 1057,
    vhpiInPortK = 1058,
    vhpiIndexedNameK = 1059,
    vhpiIntLiteralK = 1060,
    vhpiIntRangeK = 1061,
    vhpiIntTypeDeclK = 1062,
    vhpiIteratorK = 1063,
    vhpiLibraryDeclK = 1064,
    DEPRECATED_vhpiLoopStmtK = 1065,
    vhpiNextStmtK = 1066,
    vhpiNullLiteralK = 1067,
    vhpiNullStmtK = 1068,
    DEPRECATED_vhpiOperatorK = 1069,
    vhpiOthersK = 1070,
    vhpiOutPortK = 1071,
    vhpiPackBodyK = 1072,
    vhpiPackDeclK = 1073,
    vhpiPackInstK = 1074,
    vhpiParamAttrNameK = 1075,
    vhpiPhysLiteralK = 1076,
    vhpiPhysRangeK = 1077,
    vhpiPhysTypeDeclK = 1078,
    vhpiPortDeclK = 1079,
    DEPRECATED_vhpiProcCallStmtK = 1080,
    vhpiProcDeclK = 1081,
```



```
        vhpiProcessStmtK = 1082,
        DEPRECATED_vhpiProtectedTypeK = 1083,
        vhpiProtectedTypeBodyK = 1084,
        vhpiProtectedTypeDeclK = 1085,
        vhpiRealLiteralK = 1086,
        vhpiRecordTypeDeclK = 1087,
        vhpiReportStmtK = 1088,
        vhpiReturnStmtK = 1089,
        vhpiRootInstK = 1090,
        vhpiSelectSigAssignStmtK = 1091,
        vhpiSelectWaveformK = 1092,
        vhpiSelectedNameK = 1093,
        vhpiSigDeclK = 1094,
        vhpiSigParamDeclK = 1095,
        vhpiSimpAttrNameK = 1096,
        vhpiSimpleSigAssignStmtK = 1097,
        vhpiSliceNameK = 1098,
        vhpiStringLiteralK = 1099,
        vhpiSubpBodyK = 1100,
        vhpiSubtypeDeclK = 1101,
        DEPRECATED_vhpiSubtypeIndicK = 1102,
        vhpiToolK = 1103,
        vhpiTransactionK = 1104,
        vhpiTypeConvK = 1105,
        DEPRECATED_vhpiUnaryExprK = 1106,
        vhpiUnitDeclK = 1107,
        vhpiUserAttrNameK = 1108,
        vhpiVarAssignStmtK = 1109,
        vhpiVarDeclK = 1110,
        vhpiVarParamDeclK = 1111,
        vhpiWaitStmtK = 1112,
        vhpiWaveformElemK = 1113,
        vhpiWhileLoopK = 1114,
        vhpiQualifiedExprK = 1115,
        vhpiUseClauseK = 1116,
        vhpiConcAssertStmtK = 1117,
        vhpiConcProcCallStmtK = 1118,
        vhpiForeverLoopK = 1119,
        vhpiSeqAssertStmtK = 1120,
        vhpiSeqProcCallStmtK = 1121,
        vhpiSeqSigAssignStmtK = 1122,
        vhpiProtectedTypeInstK = 1123
#ifdef VHPIEXTEND_CLASSES
        VHPIEXTEND_CLASSES
#endif
        } vhpiClassKindT;

/********* methods used to traverse 1 to 1 relationships **************/
typedef enum {
        vhpiAbstractLiteral = 1301,
        vhpiActual = 1302,
        vhpiAll = 1303,
        vhpiAttrDecl = 1304,
        vhpiAttrSpec = 1305,
```

```
    vhpiBaseType = 1306,
    vhpiBaseUnit = 1307,
    DEPRECATED_vhpiBasicSignal = 1308,
    vhpiBlockConfig = 1309,
    vhpiCaseExpr = 1310,
    vhpiCondExpr = 1311,
    vhpiConfigDecl = 1312,
    vhpiConfigSpec = 1313,
    vhpiConstraint = 1314,
    vhpiContributor = 1315,
    vhpiCurCallback = 1316,
    DEPRECATED_vhpiCurEqProcess = 1317,
    vhpiCurStackFrame = 1318,
    vhpiDerefObj = 1319,
    DEPRECATED_vhpiDecl = 1320,
    vhpiDesignUnit = 1321,
    vhpiDownStack = 1322,
    DEPRECATED_vhpiElemSubtype = 1323,
    vhpiEntityAspect = 1324,
    vhpiEntityDecl = 1325,
    vhpiEqProcessStmt = 1326,
    vhpiExpr = 1327,
    vhpiFormal = 1328,
    vhpiFuncDecl = 1329,
    vhpiGroupTempDecl = 1330,
    vhpiGuardExpr = 1331,
    vhpiGuardSig = 1332,
    vhpiImmRegion = 1333,
    vhpiInPort = 1334,
    vhpiInitExpr = 1335,
    DEPRECATED_vhpiIterScheme = 1336,
    vhpiLeftExpr = 1337,
    vhpiLexicalScope = 1338,
    vhpiLhsExpr = 1339,
    vhpiLocal = 1340,
    vhpiLogicalExpr = 1341,
    DEPRECATED_vhpiName = 1342,
    DEPRECATED_vhpiOperator = 1343,
    vhpiOthers = 1344,
    vhpiOutPort = 1345,
    vhpiParamDecl = 1346,
    DEPRECATED_vhpiParamExpr = 1347,
    vhpiParent = 1348,
    vhpiPhysLiteral = 1349,
    vhpiPrefix = 1350,
    vhpiPrimaryUnit = 1351,
    vhpiProtectedTypeBody = 1352,
    vhpiProtectedTypeDecl = 1353,
    vhpiRejectTime = 1354,
    vhpiReportExpr = 1355,
    vhpiResolFunc = 1356,
    vhpiReturnExpr = 1357,
    DEPRECATED_vhpiReturnTypeMark = 1358,
    vhpiRhsExpr = 1359,
```



```
        vhpiRightExpr = 1360,
        vhpiRootInst = 1361,
        vhpiSelectExpr = 1362,
        vhpiSeverityExpr = 1363,
        vhpiSimpleName = 1364,
        vhpiSubpBody = 1365,
        vhpiSubpDecl = 1366,
        DEPRECATED_vhpiSubtype = 1367,
        vhpiSuffix = 1368,
        vhpiTimeExpr = 1369,
        vhpiTimeOutExpr = 1370,
        vhpiTool = 1371,
        vhpiType = 1372,
        DEPRECATED_vhpiTypeMark = 1373,
        vhpiUnitDecl = 1374,
        vhpiUpStack = 1375,
        vhpiUpperRegion = 1376,
        vhpiUse = 1377,
        vhpiValExpr = 1378,
        DEPRECATED_vhpiValSubtype = 1379,
        vhpiElemType = 1380,
        vhpiFirstNamedType = 1381,
        vhpiReturnType = 1382,
        vhpiValType = 1383,
        vhpiCurRegion = 1384,
        vhpiSignal = 1385,
        vhpiLibraryDecl = 1386,
        vhpiSimNet = 1387,
        vhpiAliasedName = 1388,
        vhpiCompDecl = 1389,
        vhpiProtectedTypeInst = 1390,
        vhpiGenIndex = 1391

#ifdef VHPIEXTEND_ONE_METHODS
        VHPIEXTEND_ONE_METHODS

#endif

} vhpiOneToOneT;

/******* methods used to traverse 1 to many relationships ************/
typedef enum {
        vhpiAliasDecls = 1501,
        vhpiArgvs = 1502,
        vhpiAttrDecls = 1503,
        vhpiAttrSpecs = 1504,
        vhpiBasicSignals = 1505,
        vhpiBlockStmts = 1506,
        vhpiBranchs = 1507,
        /* 1508 */
        vhpiChoices = 1509,
        vhpiCompInstStmts = 1510,
        DEPRECATED_vhpiCondExprs = 1511,
        vhpiCondWaveforms = 1512,
```

```
        vhpiConfigItems = 1513,
        vhpiConfigSpecs = 1514,
        vhpiConstDecls = 1515,
        vhpiConstraints = 1516,
        DEPRECATED_vhpiContributors = 1517,
        /* 1518 */
        vhpiDecls = 1519,
        vhpiDepUnits = 1520,
        vhpiDesignUnits = 1521,
        vhpiDrivenSigs = 1522,
        vhpiDrivers = 1523,
        vhpiElemAssocs = 1524,
        DEPRECATED_vhpiEntityClassEntrys = 1525,
        vhpiEntityDesignators = 1526,
        vhpiEnumLiterals = 1527,
        vhpiForeignfs = 1528,
        vhpiGenericAssocs = 1529,
        vhpiGenericDecls = 1530,
        vhpiIndexExprs = 1531,
        vhpiIndexedNames = 1532,
        vhpiInternalRegions = 1533,
        vhpiMembers = 1534,
        vhpiPackInsts = 1535,
        vhpiParamAssocs = 1536,
        vhpiParamDecls = 1537,
        vhpiPortAssocs = 1538,
        vhpiPortDecls = 1539,
        vhpiRecordElems = 1540,
        vhpiSelectWaveforms = 1541,
        vhpiSelectedNames = 1542,
        DEPRECATED_vhpiSensitivitys = 1543,
        vhpiSeqStmts = 1544,
        vhpiSigAttrs = 1545,
        vhpiSigDecls = 1546,
        vhpiSigNames = 1547,
        vhpiSignals = 1548,
        DEPRECATED_vhpiSpecNames = 1549,
        vhpiSpecs = 1550,
        vhpiStmts = 1551,
        vhpiTransactions = 1552,
        DEPRECATED_vhpiTypeMarks = 1553,
        vhpiUnitDecls = 1554,
        vhpiUses = 1555,
        vhpiVarDecls = 1556,
        vhpiWaveformElems = 1557,
        vhpiLibraryDecls = 1558,
        vhpiLocalLoads = 1559,
        vhpiOptimizedLoads = 1560,
        vhpiTypes = 1561,
        vhpiUseClauses = 1562,
        vhpiGenerateStmts = 1563,
        vhpiLocalContributors = 1564,
        vhpiOptimizedContributors = 1565,
        vhpiParamExprs = 1566,
```



```
        vhpiEqProcessStmts = 1567,
        vhpiEntityClassEntries = 1568,
        vhpiSensitivities = 1569

#ifdef VHPIEXTEND_MANY_METHODS
        VHPIEXTEND_MANY_METHODS
#endif

} vhpiOneToManyT;

/* Note: The following macro is defined for compatibility with
   prototype implementations that use the incorrectly spelled
   enumeration value. The macro is deprecated and will be removed
   in a future revision of the standard.
*/
#define vhpiSensitivitys DEPRECATED_vhpiSensitivitys


/****************** PROPERTIES ******************/
/******* INTEGER or BOOLEAN PROPERTIES **********/
typedef enum {
        vhpiAccessP = 1001,
        vhpiArgcP = 1002,
        vhpiAttrKindP = 1003,
        vhpiBaseIndexP = 1004,
        vhpiBeginLineNoP = 1005,
        vhpiEndLineNoP = 1006,
        vhpiEntityClassP = 1007,
        vhpiForeignKindP = 1008,
        vhpiFrameLevelP = 1009,
        vhpiGenerateIndexP = 1010,
        vhpiIntValP = 1011,
        vhpiIsAnonymousP = 1012,
        vhpiIsBasicP = 1013,
        vhpiIsCompositeP = 1014,
        vhpiIsDefaultP = 1015,
        vhpiIsDeferredP = 1016,
        vhpiIsDiscreteP = 1017,
        vhpiIsForcedP = 1018,
        vhpiIsForeignP = 1019,
        vhpiIsGuardedP = 1020,
        vhpiIsImplicitDeclP = 1021,
        DEPRECATED_vhpiIsInvalidP = 1022,
        vhpiIsLocalP = 1023,
        vhpiIsNamedP = 1024,
        vhpiIsNullP = 1025,
        vhpiIsOpenP = 1026,
        vhpiIsPLIP = 1027,
        vhpiIsPassiveP = 1028,
        vhpiIsPostponedP = 1029,
        vhpiIsProtectedTypeP = 1030,
        vhpiIsPureP = 1031,
        vhpiIsResolvedP = 1032,
        vhpiIsScalarP = 1033,
```

```
        vhpiIsSeqStmtP = 1034,
        vhpiIsSharedP = 1035,
        vhpiIsTransportP = 1036,
        vhpiIsUnaffectedP = 1037,
        vhpiIsUnconstrainedP = 1038,
        vhpiIsUninstantiatedP = 1039,
        vhpiIsUpP = 1040,
        vhpiIsVitalP = 1041,
        vhpiIteratorTypeP = 1042,
        vhpiKindP = 1043,
        vhpiLeftBoundP = 1044,
        DEPRECATED_vhpiLevelP = 1045,
        vhpiLineNoP = 1046,
        vhpiLineOffsetP = 1047,
        vhpiLoopIndexP = 1048,
        vhpiModeP = 1049,
        vhpiNumDimensionsP = 1050,
        DEPRECATED_vhpiNumFieldsP = 1051,
        vhpiNumGensP = 1052,
        vhpiNumLiteralsP = 1053,
        vhpiNumMembersP = 1054,
        vhpiNumParamsP = 1055,
        vhpiNumPortsP = 1056,
        vhpiOpenModeP = 1057,
        vhpiPhaseP = 1058,
        vhpiPositionP = 1059,
        vhpiPredefAttrP = 1060,
        /* 1061 */
        vhpiReasonP = 1062,
        vhpiRightBoundP = 1063,
        vhpiSigKindP = 1064,
        vhpiSizeP = 1065,
        vhpiStartLineNoP = 1066,
        vhpiStateP = 1067,
        vhpiStaticnessP = 1068,
        vhpiVHDLversionP = 1069,
        vhpiIdP = 1070,
        vhpiCapabilitiesP = 1071,
        vhpiAutomaticRestoreP = 1072,
        vhpiCompInstKindP = 1073,
        vhpiIsBuiltInP = 1074,
        vhpiIsDynamicP = 1075,
        vhpiIsOperatorP = 1076,
        vhpiNumFieldsP = 1077

#ifdef VHPIEXTEND_INT_PROPERTIES
        VHPIEXTEND_INT_PROPERTIES

#endif

} vhpiIntPropertyT;

/******* STRING PROPERTIES **********/
typedef enum {
```



```
        vhpiCaseNameP = 1301,
        vhpiCompNameP = 1302,
        vhpiDefNameP = 1303,
        vhpiFileNameP = 1304,
        vhpiFullCaseNameP = 1305,
        vhpiFullNameP = 1306,
        vhpiKindStrP = 1307,
        vhpiLabelNameP = 1308,
        vhpiLibLogicalNameP = 1309,
        vhpiLibPhysicalNameP = 1310,
        vhpiLogicalNameP = 1311,
        vhpiLoopLabelNameP = 1312,
        vhpiNameP = 1313,
        DEPRECATED_vhpiOpNameP = 1314,
        vhpiStrValP = 1315,
        vhpiToolVersionP = 1316,
        vhpiUnitNameP = 1317,
        vhpiSaveRestartLocationP = 1318,
        vhpiCompInstNameP = 1319,
        vhpiInstNamesP = 1320,
        vhpiSignatureNameP = 1321,
        vhpiSpecNameP = 1322

#ifdef VHPIEXTEND_STR_PROPERTIES
        VHPIEXTEND_STR_PROPERTIES

#endif
} vhpiStrPropertyT;

/******* REAL PROPERTIES **********/
typedef enum {
        vhpiFloatLeftBoundP = 1601,
        vhpiFloatRightBoundP = 1602,
        vhpiRealValP = 1603

#ifdef VHPIEXTEND_REAL_PROPERTIES
        VHPIEXTEND_REAL_PROPERTIES
#endif

} vhpiRealPropertyT;

/******* PHYSICAL PROPERTIES **********/
typedef enum {
        vhpiPhysLeftBoundP = 1651,
        vhpiPhysPositionP = 1652,
        vhpiPhysRightBoundP = 1653,
        vhpiPhysValP = 1654,
        DEPRECATED_vhpiPrecisionP = 1655,
        DEPRECATED_vhpiSimTimeUnitP = 1656,
        vhpiResolutionLimitP = 1657,
        vhpiTimeP = 1658

#ifdef VHPIEXTEND_PHYS_PROPERTIES
        VHPIEXTEND_PHYS_PROPERTIES
```

```
#endif

} vhpiPhysPropertyT;

/****************** PROPERTY VALUES ***********************/

/* vhpiCapabilitiesP */
typedef enum {
    vhpiProvidesHierarchy            = 1,
    vhpiProvidesStaticAccess         = 2,
    vhpiProvidesConnectivity         = 4,
    vhpiProvidesPostAnalysis         = 8,
    vhpiProvidesForeignModel         = 16,
    vhpiProvidesAdvancedForeignModel  = 32,
    vhpiProvidesSaveRestart          = 64,
    vhpiProvidesReset                = 128,
    vhpiProvidesDebugRuntime         = 256,
    vhpiProvidesAdvancedDebugRuntime  = 512,
    vhpiProvidesDynamicElab          = 1024

} vhpiCapabibilityT;


/* vhpiOpenModeP */
typedef enum {
        vhpiInOpen         =       1001,
        vhpiOutOpen        =       1002,
        vhpiReadOpen       =       1003,
        vhpiWriteOpen      =       1004,
        vhpiAppendOpen     =       1005
} vhpiOpenModeT;

/* vhpiModeP */
typedef enum {
        vhpiInMode         =       1001,
        vhpiOutMode        =       1002,
        vhpiInoutMode      =       1003,
        vhpiBufferMode     =       1004,
        vhpiLinkageMode    =       1005
} vhpiModeT;

/* vhpiSigKindP */
typedef enum {
        vhpiRegister       =       1001,
        vhpiBus            =       1002,
        vhpiNormal         =       1003
} vhpiSigKindT;

/* vhpiStaticnessP */
typedef enum {
        vhpiLocallyStatic  =       1001,
        vhpiGloballyStatic =       1002,
        vhpiDynamic        =       1003
} vhpiStaticnessT;
```

```
/* vhpiPredefAttrP */
typedef enum {
        vhpiActivePA       =       1001,
        vhpiAscendingPA    =       1002,
        vhpiBasePA         =       1003,
        vhpiDelayedPA      =       1004,
        vhpiDrivingPA      =       1005,
        vhpiDriving_valuePA=       1006,
        vhpiEventPA        =       1007,
        vhpiHighPA         =       1008,
        vhpiImagePA        =       1009,
        vhpiInstance_namePA=       1010,
        vhpiLast_activePA  =       1011,
        vhpiLast_eventPA   =       1012,
        vhpiLast_valuePA   =       1013,
        vhpiLeftPA         =       1014,
        vhpiLeftofPA       =       1015,
        vhpiLengthPA       =       1016,
        vhpiLowPA          =       1017,
        vhpiPath_namePA    =       1018,
        vhpiPosPA          =       1019,
        vhpiPredPA         =       1020,
        vhpiQuietPA        =       1021,
        vhpiRangePA        =       1022,
        vhpiReverse_rangePA=       1023,
        vhpiRightPA        =       1024,
        vhpiRightofPA      =       1025,
        vhpiSimple_namePA  =       1026,
        vhpiStablePA       =       1027,
        vhpiSuccPA         =       1028,
        vhpiTransactionPA  =       1029,
        vhpiValPA          =       1030,
        vhpiValuePA        =       1031
} vhpiPredefAttrT;

/* vhpiAttrKindP */
typedef enum {
  vhpiFunctionAK    = 1,
  vhpiRangeAK       = 2,
  vhpiSignalAK      = 3,
  vhpiTypeAK        = 4,
  vhpiValueAK       = 5
#ifdef VHPIEXTEND_ATTR
  VHPIEXTEND_ATTR
#endif
} vhpiAttrKindT;

/* vhpiEntityClassP */
typedef enum {
        vhpiEntityEC         =     1001,
        vhpiArchitectureEC   =     1002,
        vhpiConfigurationEC  =     1003,
        vhpiProcedureEC      =     1004,
```

```
        vhpiFunctionEC       =     1005,
        vhpiPackageEC        =     1006,
        vhpiTypeEC           =     1007,
        vhpiSubtypeEC        =     1008,
        vhpiConstantEC       =     1009,
        vhpiSignalEC         =     1010,
        vhpiVariableEC       =     1011,
        vhpiComponentEC      =     1012,
        vhpiLabelEC          =     1013,
        vhpiLiteralEC        =     1014,
        vhpiUnitsEC          =     1015,
        vhpiFileEC           =     1016,
        vhpiGroupEC          =     1017
} vhpiEntityClassT;

/* vhpiAccessP */
typedef enum {
        vhpiRead             =     1,
        vhpiWrite            =     2,
        vhpiConnectivity     =     4,
        vhpiNoAccess         =     8
} vhpiAccessT;

/* value for vhpiStateP property for callbacks */
typedef enum {
        vhpiEnable,
        vhpiDisable,
        vhpiMature /* callback has occurred */
} vhpiStateT;
/* enumeration type for vhpiCompInstKindP property */
typedef enum {
        vhpiDirect,
        vhpiComp,
        vhpiConfig
} vhpiCompInstKindT;


/* the following values are used only for the
   vhpiResolutionLimitP property and for setting the unit field
   of the value structure; they represent the physical position
   of a given VHDL time unit */
/* time unit physical position values {high, low} */
PLI_VEXTERN PLI_DLLISPEC const vhpiPhysT  vhpiFS;
PLI_VEXTERN PLI_DLLISPEC const vhpiPhysT  vhpiPS;
PLI_VEXTERN PLI_DLLISPEC const vhpiPhysT  vhpiNS;
PLI_VEXTERN PLI_DLLISPEC const vhpiPhysT  vhpiUS;
PLI_VEXTERN PLI_DLLISPEC const vhpiPhysT  vhpiMS;
PLI_VEXTERN PLI_DLLISPEC const vhpiPhysT  vhpiS;
PLI_VEXTERN PLI_DLLISPEC const vhpiPhysT  vhpiMN;
PLI_VEXTERN PLI_DLLISPEC const vhpiPhysT  vhpiHR;

/* IEEE std_logic values */
#define vhpiU                   0   /* uninitialized */
#define vhpiX                   1   /* unknown */
```



```
#define vhpi0                   2   /* forcing 0 */
#define vhpi1                   3   /* forcing 1 */
#define vhpiZ                   4   /* high impedance */
#define vhpiW                   5   /* weak unknown */
#define vhpiL                   6   /* weak 0 */
#define vhpiH                   7   /* weak 1 */
#define vhpiDontCare            8   /* don't care */

/* IEEE std bit values */
#define vhpibit0                0   /* bit 0 */
#define vhpibit1                1   /* bit 1 */

/* IEEE std boolean values */
#define vhpiFalse               0   /* false */
#define vhpiTrue                1   /* true */

/************** vhpiPhaseP property values *************/
typedef enum {
       vhpiRegistrationPhase   = 1,
       vhpiAnalysisPhase       = 2,
       vhpiElaborationPhase    = 3,
       vhpiInitializationPhase = 4,
       vhpiSimulationPhase     = 5,
       vhpiTerminationPhase    = 6,
       vhpiSavePhase           = 7,
       vhpiRestartPhase        = 8,
       vhpiResetPhase          = 9
} vhpiPhaseT ;

/*************** PLI error information structure ***************/

typedef enum {
        vhpiNote        = 1,
        vhpiWarning     = 2,
        vhpiError       = 3,
        vhpiFailure     = 6,
        vhpiSystem      = 4,
        vhpiInternal    = 5
} vhpiSeverityT;

typedef struct vhpiErrorInfoS
{
  vhpiSeverityT    severity;
  char       *message;
  char       *str;
  char       *file; /* Name of the VHDL file where the VHPI error
                       originated */
  int32_t     line; /* Line number in the VHDL file */
} vhpiErrorInfoT;

/******************** callback structures ***********************/
/* callback user data structure */

typedef struct vhpiCbDataS
```

```
{
  int32_t reason;            /* callback reason */
  void (*cb_rtn) (const struct vhpiCbDataS *);  /* call routine */
  vhpiHandleT obj;            /* trigger object */
  vhpiTimeT *time;            /* callback time */
  vhpiValueT *value;          /* trigger object value */
  void *user_data;            /* pointer to user data to be passed
                                 to the callback function */
} vhpiCbDataT;

/************************ CALLBACK REASONS ************************/
/*********************** Simulation object related ****************/
/* These are repetitive callbacks */
#define vhpiCbValueChange          1001
#define vhpiCbForce                1002
#define vhpiCbRelease              1003
#define vhpiCbTransaction          1004 /* optional callback reason */

/************************** Statement related **********************/
/* These are repetitive callbacks */
#define vhpiCbStmt                 1005
#define vhpiCbResume               1006
#define vhpiCbSuspend              1007
#define vhpiCbStartOfSubpCall      1008
#define vhpiCbEndOfSubpCall        1009

/************************** Time related ***************************/
/* the Rep callback reasons are the repeated versions
   of the callbacks */

#define vhpiCbAfterDelay           1010
#define vhpiCbRepAfterDelay        1011

/*********************** Simulation cycle phase related ************/
#define vhpiCbNextTimeStep         1012
#define vhpiCbRepNextTimeStep      1013
#define vhpiCbStartOfNextCycle     1014
#define vhpiCbRepStartOfNextCycle  1015
#define vhpiCbStartOfProcesses     1016
#define vhpiCbRepStartOfProcesses  1017
#define vhpiCbEndOfProcesses       1018
#define vhpiCbRepEndOfProcesses    1019
#define vhpiCbLastKnownDeltaCycle  1020
#define vhpiCbRepLastKnownDeltaCycle 1021
#define vhpiCbStartOfPostponed     1022
#define vhpiCbRepStartOfPostponed  1023
#define vhpiCbEndOfTimeStep        1024
#define vhpiCbRepEndOfTimeStep     1025

/************************** Action related *************************/
/* these are one time callback unless otherwise noted */
#define vhpiCbStartOfTool          1026
#define vhpiCbEndOfTool            1027
#define vhpiCbStartOfAnalysis      1028
```

```
#define vhpiCbEndOfAnalysis        1029
#define vhpiCbStartOfElaboration   1030
#define vhpiCbEndOfElaboration     1031
#define vhpiCbStartOfInitialization 1032
#define vhpiCbEndOfInitialization  1033
#define vhpiCbStartOfSimulation    1034
#define vhpiCbEndOfSimulation      1035
#define vhpiCbQuiescense           1036 /* repetitive */
#define vhpiCbPLIError             1037 /* repetitive */
#define vhpiCbStartOfSave          1038
#define vhpiCbEndOfSave            1039
#define vhpiCbStartOfRestart       1040
#define vhpiCbEndOfRestart         1041
#define vhpiCbStartOfReset         1042
#define vhpiCbEndOfReset           1043
#define vhpiCbEnterInteractive     1044 /* repetitive */
#define vhpiCbExitInteractive      1045 /* repetitive */
#define vhpiCbSigInterrupt         1046 /* repetitive */

/* Foreign model callbacks */
#define vhpiCbTimeOut              1047 /* non repetitive */
#define vhpiCbRepTimeOut           1048 /* repetitive */
#define vhpiCbSensitivity          1049 /* repetitive */

/************************ CALLBACK FLAGS ***************************/
#define vhpiReturnCb  0x00000001
#define vhpiDisableCb 0x00000010

/************** vhpiAutomaticRestoreP property values *************/
typedef enum {
       vhpiRestoreAll       = 1,
       vhpiRestoreUserData  = 2,
       vhpiRestoreHandles   = 4,
       vhpiRestoreCallbacks = 8
} vhpiAutomaticRestoreT ;


/******************** FUNCTION DECLARATIONS ********************/

XXTERN int vhpi_assert (vhpiSeverityT severity,
                        char *formatmsg,
                        ...);

/* callback related */

XXTERN vhpiHandleT vhpi_register_cb (vhpiCbDataT *cb_data_p,
                                     int32_t flags);

XXTERN int vhpi_remove_cb (vhpiHandleT cb_obj);

XXTERN int vhpi_disable_cb (vhpiHandleT cb_obj);

XXTERN int vhpi_enable_cb (vhpiHandleT cb_obj);
```

```
XXTERN int vhpi_get_cb_info (vhpiHandleT object,
                             vhpiCbDataT *cb_data_p);

/* utilities for sensitivity-set bitmaps */
/* The replacement text for these macros is implementation defined */
/* The behavior is specified in G.1 */
#define VHPI_SENS_ZERO(sens)        vhpi_sens_zero(sens)
#define VHPI_SENS_SET(obj, sens)    vhpi_sens_set(obj, sens)
#define VHPI_SENS_CLR(obj, sens)    vhpi_sens_clr(obj, sens)
#define VHPI_SENS_ISSET(obj, sens)  vhpi_sens_isset(obj, sens)
#define VHPI_SENS_FIRST(sens)       vhpi_sens_first(sens)

/* for obtaining handles */

XXTERN vhpiHandleT vhpi_handle_by_name (const char *name,
                                        vhpiHandleT scope);

XXTERN vhpiHandleT vhpi_handle_by_index (vhpiOneToManyT itRel,
                                         vhpiHandleT parent,
                                         int32_t indx);

/* for traversing relationships */

XXTERN vhpiHandleT vhpi_handle (vhpiOneToOneT type,
                                vhpiHandleT referenceHandle);

XXTERN vhpiHandleT vhpi_iterator (vhpiOneToManyT type,
                                  vhpiHandleT referenceHandle);

XXTERN vhpiHandleT vhpi_scan (vhpiHandleT iterator);

/* for processsing properties */

XXTERN vhpiIntT vhpi_get (vhpiIntPropertyT property,
                          vhpiHandleT object);

XXTERN const vhpiCharT * vhpi_get_str (vhpiStrPropertyT property,
                                       vhpiHandleT object);

XXTERN vhpiRealT vhpi_get_real (vhpiRealPropertyT property,
                                vhpiHandleT object);

XXTERN vhpiPhysT vhpi_get_phys (vhpiPhysPropertyT property,
                                vhpiHandleT object);

/* for access to protected types */

typedef int (*vhpiUserFctT)();

XXTERN int vhpi_protected_call (vhpiHandleT varHdl,
                                vhpiUserFctT userFct,
                                void *userData);

/* value processing */
```

```
/* vhpi_put_value modes */
typedef enum {
  vhpiDeposit,
  vhpiDepositPropagate,
  vhpiForce,
  vhpiForcePropagate,
  vhpiRelease,
  vhpiSizeConstraint
} vhpiPutValueModeT;

typedef enum {
  vhpiInertial,
  vhpiTransport
} vhpiDelayModeT;

XXTERN int vhpi_get_value (vhpiHandleT expr,
                           vhpiValueT *value_p);

XXTERN int vhpi_put_value (vhpiHandleT object,
                           vhpiValueT *value_p,
                           vhpiPutValueModeT mode);

XXTERN int vhpi_schedule_transaction (vhpiHandleT drivHdl,
                                      vhpiValueT *value_p,
                                      uint32_t numValues,
                                      vhpiTimeT *delayp,
                                      vhpiDelayModeT delayMode,
                                      vhpiTimeT *pulseRejp);

XXTERN int vhpi_format_value (const vhpiValueT *in_value_p,
                              vhpiValueT *out_value_p);

/* time processing */

XXTERN void vhpi_get_time (vhpiTimeT *time_p,
                           long *cycles);

#define vhpiNoActivity -1

XXTERN int vhpi_get_next_time (vhpiTimeT *time_p);

/* simulation control */

typedef enum {
  vhpiStop     = 0,
  vhpiFinish   = 1,
  vhpiReset    = 2
#ifdef VHPIEXTEND_CONTROL
  VHPIEXTEND_CONTROL
#endif
} vhpiSimControlT;

XXTERN int vhpi_control (vhpiSimControlT command,
```

```
                          ...);

/* I/O routine */

XXTERN int vhpi_printf (const char *format,
                                                ...);
XXTERN int vhpi_vprintf (const char *format, va_list args);

/* utilities to print VHDL strings */

XXTERN int vhpi_is_printable( char ch );


/* utility routines */

XXTERN int vhpi_compare_handles (vhpiHandleT handle1,
                                 vhpiHandleT handle2);

XXTERN int vhpi_check_error (vhpiErrorInfoT *error_info_p);

XXTERN int vhpi_release_handle (vhpiHandleT object);

/* creation functions */

XXTERN vhpiHandleT vhpi_create (vhpiClassKindT kind,
                                vhpiHandleT handle1,
                                vhpiHandleT handle2);

/* Foreign model data structures and functions */

typedef enum {
   vhpiArchF   = 1,
   vhpiFuncF   = 2,
   vhpiProcF   = 3,
   vhpiLibF    = 4,
   vhpiAppF    = 5
} vhpiForeignKindT;

typedef struct vhpiForeignDataS {
    vhpiForeignKindT kind;
    char * libraryName;
    char * modelName;
    void (*elabf)(const struct vhpiCbDataS *cb_data_p);
    void (*execf)(const struct vhpiCbDataS *cb_data_p);
} vhpiForeignDataT;

XXTERN vhpiHandleT vhpi_register_foreignf
               (vhpiForeignDataT *foreignDatap);

/* vhpi_get_foreign_info is DEPRECATED and is replaced
   by the function vhpi_get_foreignf_info */
XXTERN int vhpi_get_foreignf_info (vhpiHandleT hdl,
                                   vhpiForeignDataT *foreignDatap);
```



```
/* for saving and restoring foreign models data */

XXTERN size_t vhpi_get_data (int32_t id,
                             void *dataLoc,
                             size_t numBytes);

XXTERN size_t vhpi_put_data (int32_t id,
                             void *dataLoc,
                             size_t numBytes);

#ifdef VHPIEXTEND_FUNCTIONS
       VHPIEXTEND_FUNCTIONS
#endif

/*************************** Typedef for VHPI registration functions
 ***************************/

typedef void (*vhpiRegistrationFctT)();


#undef PLI_EXTERN
#undef PLI_VEXTERN

#ifdef VHPI_USER_DEFINED_DLLISPEC
#undef VHPI_USER_DEFINED_DLLISPEC
#undef PLI_DLLISPEC
#endif
#ifdef VHPI_USER_DEFINED_DLLESPEC
#undef VHPI_USER_DEFINED_DLLESPEC
#undef PLI_DLLESPEC
#endif

#ifdef PLI_PROTOTYPES
#undef PLI_PROTOTYPES
#undef XXTERN
#undef EETERN
#endif

#ifdef  __cplusplus
}
#endif

#endif /* VHPI_USER_H */
```

B.2 Macros for sensitivity-set bitmaps

B.2.1 General

The macros for manipulating sensitivity-set bitmaps, defined in the header file, are described in this subclause (B.2).

The definitions of the macros in the header file invoke functions defined in the file `vhpi_sens.c` (see A.4.4). A tool provider may replace the definitions with implementation-specific definitions that have the effect described in this subclause (B.2). Such definitions may invoke implementation-defined functions or may be in the form of in-line code.

B.2.2 VHPI_SENS_ZERO

Clears a sensitivity-set bitmap.

Synopsis:

```
VHPI_SENS_ZERO(sens)
```

Description:

The argument `sens` is a pointer to a sensitivity-set bitmap. The macro clears all of the bits in the sensitivity-set bitmap to 0.

B.2.3 VHPI_SENS_SET

Sets a bit in a sensitivity-set bitmap.

Synopsis:

```
VHPI_SENS_SET(obj, sens)
```

Description:

The argument `obj` is an integer representing the index of a signal in a sensitivity set, and the argument `sens` is a pointer to a sensitivity-set bitmap. The macro sets to 1 the bit in the sensitivity-set bitmap corresponding to the signal with the given index.

B.2.4 VHPI_SENS_CLR

Clears a bit in a sensitivity-set bitmap.

Synopsis:

```
VHPI_SENS_CLR(obj, sens)
```

Description:

The argument `obj` is an integer representing the index of a signal in a sensitivity set, and the argument `sens` is a pointer to a sensitivity-set bitmap. The macro clears to 0 the bit in the sensitivity-set bitmap corresponding to the signal with the given index.

B.2.5 VHPI_SENS_ISSET

Determines whether a specific bit in a sensitivity-set bitmap is set.

Synopsis:

```
VHPI_SENS_ISSET(obj, sens)
```

Description:

The argument `obj` is an integer representing the index of a signal in a sensitivity set, and the argument `sens` is a pointer to a sensitivity-set bitmap. The macro yields an integer that is the value of the bit in the sensitivity-set bitmap corresponding to the signal with the given index.

B.2.6 VHPI_SENS_FIRST

Determines whether any bit in a sensitivity-set bitmap is set.

Synopsis:

```
VHPI_SENS_FIRST(sens)
```

Description:

The argument `sens` is a pointer to a sensitivity-set bitmap. If any of the bits in the sensitivity-set bitmap corresponding to signals in a sensitivity set is 1, the macro yields an integer that is the least index of the signals for which the corresponding bit is set. Otherwise, the macro yields the value `vhpiUndefined`.

B.3 Implementation-specific extensions

A tool provider may provide implementation-defined functionality in addition to that described by this standard. Where such functionality requires declarations in the `vhpi_user.h` header file, those declarations shall be provided by definitions of the following macros:

`VHPIEXTEND_VAL_FORMATS`	Enumeration constants for implementation-defined value formats.
`VHPIEXTEND_CLASSES`	Enumeration constants for implementation-defined classes.
`VHPIEXTEND_ONE_METHODS`	Enumeration constants for implementation-defined one-to-one associations.
`VHPIEXTEND_MANY_METHODS`	Enumeration constants for implementation-defined one-to-many associations.
`VHPIEXTEND_INT_PROPERTIES`	Enumeration constants for implementation-defined integer properties.
`VHPIEXTEND_STR_PROPERTIES`	Enumeration constants for implementation-defined string properties.
`VHPIEXTEND_REAL_PROPERTIES`	Enumeration constants for implementation-defined real properties.
`VHPIEXTEND_PHYS_PROPERTIES`	Enumeration constants for implementation-defined physical properties.
`VHPIEXTEND_ATTR`	Enumeration constants for implementation-defined attribute kinds.
`VHPIEXTEND_CONTROL`	Enumeration constants for implementation-defined control actions.
`VHPIEXTEND_FUNCTIONS`	Prototypes for implementation-defined functions.

The macros shall be defined before compilation of the `vhpi_user.h` file and shall be defined in such a way that their instantiation in the `vhpi_user.h` file results in legal C declarations.

The range of enumeration values from 1000 to 2000, inclusive, of enumeration constants of types `vhpiClassKindT`, `vhpiOneToOneT`, `vhpiOneToManyT`, `vhpiIntPropertyT`, `vhpiStrPropertyT`, `vhpiRealPropertyT`, and `vhpiPhysPropertyT` are reserved and shall not be used for implementation defined functionality.

Annex C

(informative)

Syntax summary

This annex provides a summary of the syntax for VHDL. Productions are ordered alphabetically by left-hand nonterminal name. The number listed to the right indicates the clause or subclause where the production is given.

absolute_pathname ::= . partial_pathname [§ **8.7**]

abstract_literal ::= decimal_literal | based_literal [§ **15.5.1**]

access_type_definition ::= **access** subtype_indication [§ **5.4.1**]

actual_designator ::= [§ **6.5.7.1**]
[**inertial**] expression
| *signal*_name
| *variable*_name
| *file*_name
| subtype_indication
| *subprogram*_name
| *instantiated_package*_name
| **open**

actual_parameter_part ::= *parameter*_association_list [§ **9.3.4**]

actual_part ::= [§ **6.5.7.1**]
actual_designator
| *function*_name (actual_designator)
| type_mark (actual_designator)

adding_operator ::= + | – | & [§ **9.2**]

aggregate ::= [§ **9.3.3.1**]
(element_association { , element_association })

alias_declaration ::= [§ **6.6.1**]
alias alias_designator [: subtype_indication] **is** name [signature] ;

alias_designator ::= identifier | character_literal | operator_symbol [§ **6.6.1**]

allocator ::= [§ **9.3.7**]
new subtype_indication
| **new** qualified_expression

architecture_body ::= [§ **3.3.1**]
architecture identifier **of** *entity*_name **is**
architecture_declarative_part
begin
architecture_statement_part

end [**architecture**] [*architecture*_simple_name] ;

architecture_declarative_part ::= [§ **3.3.2**]
{ block_declarative_item }

architecture_statement_part ::= [§ **3.3.3**]
{ concurrent_statement }

array_constraint ::= [§ **5.3.2.1**]
index_constraint [array_element_constraint]
| (**open**) [array_element_constraint]

array_element_constraint ::= element_constraint [§ **5.3.2.1**]

array_element_resolution ::= resolution_indication [§ **6.3**]

array_type_definition ::= [§ **5.3.2.1**]
unbounded_array_definition | constrained_array_definition

assertion ::= [§ **10.3**]
assert condition
[**report** expression]
[**severity** expression]

assertion_statement ::= [label :] assertion ; [§ **10.3**]

association_element ::= [§ **6.5.7.1**]
[formal_part =>] actual_part

association_list ::= [§ **6.5.7.1**]
association_element { , association_element }

attribute_declaration ::= [§ **6.7**]
attribute identifier : type_mark ;

attribute_designator ::= *attribute*_simple_name [§ **8.6**]

attribute_name ::= [§ **8.6**]
prefix [signature] ' attribute_designator [(expression)]

attribute_specification ::= [§ **7.2**]
attribute attribute_designator **of** entity_specification **is** expression ;

base ::= integer [§ **15.5.3**]

base_specifier ::= B | O | X | UB | UO | UX | SB | SO | SX | D [§ **15.8**]

based_integer ::= [§ **15.5.3**]
extended_digit { [underline] extended_digit }

based_literal ::= [§ **15.5.3**]
base # based_integer [. based_integer] # [exponent]

basic_character ::= [§ **15.2**]

basic_graphic_character | format_effector

basic_graphic_character ::= [§ **15.2**]
upper_case_letter | digit | special_character| space_character

basic_identifier ::= letter { [underline] letter_or_digit } [§ **15.4.2**]

binding_indication ::= [§ **7.3.2.1**]
[**use** entity_aspect]
[generic_map_aspect]
[port_map_aspect]

bit_string_literal ::= [integer] base_specifier " [bit_value] " [§ **15.8**]

bit_value ::= graphic_character { [underline] graphic_character } [§ **15.8**]

block_configuration ::= [§ **3.4.2**]
for block_specification
{ use_clause }
{ configuration_item }
end for ;

block_declarative_item ::= [§ **3.3.2**]
subprogram_declaration
| subprogram_body
| subprogram_instantiation_declaration
| package_declaration
| package_body
| package_instantiation_declaration
| type_declaration
| subtype_declaration
| constant_declaration
| signal_declaration
| *shared*_variable_declaration
| file_declaration
| alias_declaration
| component_declaration
| attribute_declaration
| attribute_specification
| configuration_specification
| disconnection_specification
| use_clause
| group_template_declaration
| group_declaration
| *PSL*_Property_Declaration
| *PSL*_Sequence_Declaration
| *PSL*_Clock_Declaration

block_declarative_part ::= [§ **11.2**]
{ block_declarative_item }

block_header ::= [§ **11.2**]
[generic_clause
[generic_map_aspect ;]]

```
    [ port_clause
    [ port_map_aspect ; ] ]

block_specification ::=                                              [§ 3.4.2]
      architecture_name
    | block_statement_label
    | generate_statement_label [ ( generate_specification ) ]

block_statement ::=                                                  [§ 11.2]
    block_label :
        block [ ( guard_condition ) ] [ is ]
            block_header
            block_declarative_part
        begin
            block_statement_part
        end block [ block_label ] ;

block_statement_part ::=                                             [§ 11.2]
    { concurrent_statement }

case_generate_alternative ::=                                        [§ 11.8]
    when [ alternative_label : ] choices =>
        generate_statement_body

case_generate_statement ::=                                          [§ 11.8]
    generate_label :
        case expression generate
            case_generate_alternative
            { case_generate_alternative }
        end generate [ generate_label ] ;

case_statement ::=                                                   [§ 10.9]
    [ case_label : ]
        case [ ? ] expression is
            case_statement_alternative
            { case_statement_alternative }
        end case [ ? ] [ case_label ] ;

case_statement_alternative ::=                                       [§ 10.9]
    when choices =>
        sequence_of_statements

character_literal ::= ' graphic_character '                          [§ 15.6]

choice ::=                                                           [§ 9.3.3.1]
      simple_expression
    | discrete_range
    | element_simple_name
    | others

choices ::= choice { | choice }                                      [§ 9.3.3.1]

component_configuration ::=                                          [§ 3.4.3]
    for component_specification
```

```
        [ binding_indication ; ]
        { verification_unit_binding_indication ; }
        [ block_configuration ]
    end for ;

component_declaration ::=                                                    [§ 6.8]
    component identifier [ is ]
        [ local_generic_clause ]
        [ local_port_clause ]
    end component [ component_simple_name ] ;

component_instantiation_statement ::=                                        [§ 11.7.1]
    instantiation_label :
        instantiated_unit
            [ generic_map_aspect ]
            [ port_map_aspect ] ;

component_specification ::=                                                  [§ 7.3.1]
    instantiation_list : component_name

composite_type_definition ::=                                                [§ 5.3.1]
      array_type_definition
    | record_type_definition

compound_configuration_specification ::=                                     [§ 7.3.1]
    for component_specification binding_indication ;
        verification_unit_binding_indication ;
        { verification_unit_binding_indication ; }
    end for ;

concurrent_assertion_statement ::=                                           [§ 11.5]
    [ label : ] [ postponed ] assertion ;

concurrent_conditional_signal_assignment ::=                                 [§ 11.6]
    target <= [ guarded ] [ delay_mechanism ] conditional_waveforms ;

concurrent_procedure_call_statement ::=                                      [§ 11.4]
    [ label : ] [ postponed ] procedure_call ;

concurrent_selected_signal_assignment ::=                                    [§ 11.6]
    with expression select [ ? ]
        target <= [ guarded ] [ delay_mechanism ] selected_waveforms ;

concurrent_signal_assignment_statement ::=                                   [§ 11.6]
      [ label : ] [ postponed ] concurrent_simple_signal_assignment
    | [ label : ] [ postponed ] concurrent_conditional_signal_assignment
    | [ label : ] [ postponed ] concurrent_selected_signal_assignment

concurrent_simple_signal_assignment ::=                                      [§ 11.6]
    target <= [ guarded ] [ delay_mechanism ] waveform ;

concurrent_statement ::=                                                     [§ 11.1]
      block_statement
    | process_statement
```

| concurrent_procedure_call_statement
| concurrent_assertion_statement
| concurrent_signal_assignment_statement
| component_instantiation_statement
| generate_statement
| *PSL*_PSL_Directive

condition ::= expression [§ **10.2**]

condition_clause ::= **until** condition [§ **10.2**]

condition_operator ::= ?? [§ **9.2.1**]

conditional_expressions ::= [§ **10.5.3**]
expression **when** condition
{ **else** expression **when** condition }
[**else** expression]

conditional_force_assignment ::= [§ **10.5.3**]
target <= **force** [force_mode] conditional_expressions ;

conditional_signal_assignment ::= [§ **10.5.3**]
conditional_waveform_assignment
| conditional_force_assignment

conditional_variable_assignment ::= [§ **10.6.3**]
target := conditional_expressions ;

conditional_waveform_assignment ::= [§ **10.5.3**]
target <= [delay_mechanism] conditional_waveforms ;

conditional_waveforms ::= [§ **10.5.3**]
waveform **when** condition
{ **else** waveform **when** condition }
[**else** waveform]

configuration_declaration ::= [§ **3.4.1**]
configuration identifier **of** *entity*_name **is**
configuration_declarative_part
{ verification_unit_binding_indication ; }
block_configuration
end [**configuration**] [*configuration*_simple_name] ;

configuration_declarative_item ::= [§ **3.4.1**]
use_clause
| attribute_specification
| group_declaration

configuration_declarative_part ::= [§ **3.4.1**]
{ configuration_declarative_item }

configuration_item ::= [§ **3.4.2**]
block_configuration
| component_configuration

configuration_specification ::= [§ **7.3.1**]
    simple_configuration_specification
  | compound_configuration_specification

constant_declaration ::= [§ **6.4.2.2**]
  constant identifier_list : subtype_indication [:= expression] ;

constrained_array_definition ::= [§ **5.3.2.1**]
  array index_constraint **of** *element*_subtype_indication

constraint ::= [§ **6.3**]
    range_constraint
  | array_constraint
  | record_constraint

context_clause ::= { context_item } [§ **13.4**]

context_declaration ::= [§ **13.3**]
  context identifier **is**
    context_clause
  end [**context**] [*context*_simple_name] ;

context_item ::= [§ **13.4**]
    library_clause
  | use_clause
  | context_reference

context_reference ::= [§ **13.4**]
  context selected_name { , selected_name } ;

decimal_literal ::= integer [. integer] [exponent] [§ **15.5.2**]

delay_mechanism ::= [§ **10.5.2.1**]
    transport
  | [**reject** *time*_expression] **inertial**

design_file ::= design_unit { design_unit } [§ **13.1**]

design_unit ::= context_clause library_unit [§ **13.1**]

designator ::= identifier | operator_symbol [§ **4.2.1**]

direction ::= **to** | **downto** [§ **5.2.1**]

disconnection_specification ::= [§ **7.4**]
  disconnect guarded_signal_specification **after** *time*_expression ;

discrete_range ::= *discrete*_subtype_indication | range [§ **5.3.2.1**]

element_association ::= [§ **9.3.3.1**]
  [choices =>] expression

element_constraint ::= [§ **6.3**]

```
    array_constraint
  | record_constraint

element_declaration ::=                                                    [§ 5.3.3]
    identifier_list : element_subtype_definition ;

element_resolution ::= array_element_resolution | record_resolution        [§ 6.3]

element_subtype_definition ::= subtype_indication                          [§ 5.3.3]

entity_aspect ::=                                                          [§ 7.3.2.2]
    entity entity_name [ ( architecture_identifier ) ]
  | configuration configuration_name
  | open

entity_class ::=                                                           [§ 7.2]
    entity
  | architecture
  | configuration
  | procedure
  | function
  | package
  | type
  | subtype
  | constant
  | signal
  | variable
  | component
  | label
  | literal
  | units
  | group
  | file
  | property
  | sequence

entity_class_entry ::= entity_class [ <> ]                                 [§ 6.9]

entity_class_entry_list ::=                                                [§ 6.9]
    entity_class_entry { , entity_class_entry }

entity_declaration ::=                                                     [§ 3.2.1]
    entity identifier is
        entity_header
        entity_declarative_part
    [ begin
        entity_statement_part ]
    end [ entity ] [ entity_simple_name ] ;

entity_declarative_item ::=                                                [§ 3.2.3]
    subprogram_declaration
  | subprogram_body
  | subprogram_instantiation_declaration
  | package_declaration
```

| package_body
| package_instantiation_declaration
| type_declaration
| subtype_declaration
| constant_declaration
| signal_declaration
| *shared*_variable_declaration
| file_declaration
| alias_declaration
| attribute_declaration
| attribute_specification
| disconnection_specification
| use_clause
| group_template_declaration
| group_declaration
| *PSL*_Property_Declaration
| *PSL*_Sequence_Declaration
| *PSL*_Clock_Declaration

entity_declarative_part ::= [§ **3.2.3**]
{ entity_declarative_item }

entity_designator ::= entity_tag [signature] [§ **7.2**]

entity_header ::= [§ **3.2.3**]
[*formal*_generic_clause]
[*formal*_port_clause]

entity_name_list ::= [§ **7.2**]
entity_designator { , entity_designator }
| **others**
| **all**

entity_specification ::= [§ **7.2**]
entity_name_list : entity_class

entity_statement ::= [§ **3.2.4**]
concurrent_assertion_statement
| *passive*_concurrent_procedure_call_statement
| *passive*_process_statement
| *PSL*_PSL_Directive

entity_statement_part ::= [§ **3.2.4**]
{ entity_statement }

entity_tag ::= simple_name | character_literal | operator_symbol [§ **7.2**]

enumeration_literal ::= identifier | character_literal [§ **5.2.2.1**]

enumeration_type_definition ::= [§ **5.2.2.1**]
(enumeration_literal { , enumeration_literal })

exit_statement ::= [§ **10.2**]
[label :] **exit** [*loop*_label] [**when** condition] ;

exponent ::= E [+] integer | E – integer [§ **15.5.2**]

expression ::= [§ **9.1**]
condition_operator primary
| logical_expression

extended_digit ::= digit | letter [§ **15.5.3**]

extended_identifier ::= \ graphic_character { graphic_character } \ [§ **15.4.3**]

external_name ::= [§ **8.7**]
external_constant_name
| external_signal_name
| external_variable_name

external_constant_name ::= [§ **8.7**]
<< **constant** external_pathname : subtype_indication >>

external_signal_name ::= [§ **8.7**]
<< **signal** external_pathname : subtype_indication >>

external_variable_name ::= [§ **8.7**]
<< **variable** external_pathname : subtype_indication >>

external_pathname ::= [§ **8.7**]
package_pathname
| absolute_pathname
| relative_pathname

factor ::= [§ **9.1**]
primary [** primary]
| **abs** primary
| **not** primary
| logical_operator primary

file_declaration ::= [§ **6.4.2.5**]
file identifier_list : subtype_indication [file_open_information] ;

file_logical_name ::= *string*_expression [§ **6.4.2.5**]

file_open_information ::= [§ **6.4.2.5**]
[**open** *file_open_kind*_expression] **is** file_logical_name

file_type_definition ::= [§ **5.5.1**]
file of type_mark

floating_type_definition ::= range_constraint [§ **5.2.5.1**]

for_generate_statement ::= [§ **11.8**]
*generate*_label :
for *generate*_parameter_specification **generate**
generate_statement_body
end generate [*generate*_label] ;

force_mode ::= **in** | **out** [§ **10.5.2.1**]

formal_designator ::= [§ **6.5.7.1**]
*generic*_name
| *port*_name
| *parameter*_name

formal_parameter_list ::= *parameter*_interface_list [§ **4.2.2.1**]

formal_part ::= [§ **6.5.7.1**]
formal_designator
| *function*_name (formal_designator)
| type_mark (formal_designator)

full_type_declaration ::= [§ **6.2**]
type identifier **is** type_definition ;

function_call ::= [§ **9.3.4**]
*function*_name [(actual_parameter_part)]

function_specification ::= [§ **4.2.1**]
[**pure** | **impure**] **function** designator
subprogram_header
[[**parameter**] (formal_parameter_list)] **return** type_mark

generate_specification ::= [§ **3.4.2**]
*static*_discrete_range
| *static*_expression
| *alternative*_label

generate_statement ::= [§ **11.8**]
for_generate_statement
| if_generate_statement
| case_generate_statement

generate_statement_body ::= [§ **11.8**]
[block_declarative_part
begin]
{ concurrent_statement }
[**end** [*alternative*_label] ;]

generic_clause ::= [§ **6.5.6.2**]
generic (generic_list) ;

generic_list ::= *generic*_interface_list [§ **6.5.6.2**]

generic_map_aspect ::= [§ **6.5.6.2**]
generic map (*generic*_association_list)

graphic_character ::= [§ **15.2**]
basic_graphic_character | lower_case_letter | other_special_character

group_constituent ::= name | character_literal [§ **6.10**]

group_constituent_list ::= group_constituent { , group_constituent } [§ **6.10**]

group_declaration ::= [§ **6.10**]
    group identifier : *group_template*_name (group_constituent_list) ;

group_template_declaration ::= [§ **6.9**]
    group identifier **is** (entity_class_entry_list) ;

guarded_signal_specification ::= [§ **7.4**]
    *guarded*_signal_list : type_mark

identifier ::= basic_identifier | extended_identifier [§ **15.4.1**]

identifier_list ::= identifier { , identifier } [§ **5.3.3**]

if_generate_statement ::= [§ **11.8**]
    *generate*_label :
        if [*alternative*_label :] condition **generate**
            generate_statement_body
        { **elsif** [*alternative*_label :] condition **generate**
            generate_statement_body }
        [**else** [*alternative*_label :] **generate**
            generate_statement_body]
        end generate [*generate*_label] ;

if_statement ::= [§ **10.8**]
    [*if*_label :]
        if condition **then**
            sequence_of_statements
        { **elsif** condition **then**
            sequence_of_statements }
        [**else**
            sequence_of_statements]
        end if [*if*_label] ;

incomplete_type_declaration ::= **type** identifier ; [§ **5.4.2**]

index_constraint ::= (discrete_range { , discrete_range }) [§ **5.3.2.1**]

index_subtype_definition ::= type_mark **range** <> [§ **5.3.2.1**]

indexed_name ::= prefix (expression { , expression }) [§ **8.4**]

instantiated_unit ::= [§ **11.7.1**]
      [**component**] *component*_name
    | **entity** *entity*_name [(*architecture*_identifier)]
    | **configuration** *configuration*_name

instantiation_list ::= [§ **7.3.1**]
      *instantiation*_label { , *instantiation*_label }
    | **others**
    | **all**

integer ::= digit { [underline] digit } [§ **15.5.2**]

integer_type_definition ::= range_constraint [§ **5.2.3.1**]

interface_constant_declaration ::= [§ **6.5.2**]
[**constant**] identifier_list : [**in**] subtype_indication [:= *static*_expression]

interface_declaration ::= [§ **6.5.1**]
interface_object_declaration
| interface_type_declaration
| interface_subprogram_declaration
| interface_package_declaration

interface_element ::= interface_declaration [§ **6.5.6.1**]

interface_file_declaration ::= [§ **6.5.2**]
file identifier_list : subtype_indication

interface_function_specification ::= [§ **6.5.4**]
[**pure** | **impure**] **function** designator
[[**parameter**] (formal_parameter_list)] **return** type_mark

interface_incomplete_type_declaration ::= **type** identifier [§ **6.5.3**]

interface_list ::= [§ **6.5.6.1**]
interface_element { ; interface_element }

interface_object_declaration ::= [§ **6.5.2**]
interface_constant_declaration
| interface_signal_declaration
| interface_variable_declaration
| interface_file_declaration

interface_package_declaration ::= [§ **6.5.5**]
package identifier **is**
new *uninstantiated_package*_name interface_package_generic_map_aspect

interface_package_generic_map_aspect ::= [§ **6.5.5**]
generic_map_aspect
| **generic map** (<>)
| **generic map** (**default**)

interface_procedure_specification ::= [§ **6.5.4**]
procedure designator
[[**parameter**] (formal_parameter_list)]

interface_signal_declaration ::= [§ **6.5.2**]
[**signal**] identifier_list : [mode] subtype_indication [**bus**] [:= *static*_expression]

interface_subprogram_declaration ::= [§ **6.5.4**]
interface_subprogram_specification [**is** interface_subprogram_default]

interface_subprogram_default ::= *subprogram*_name | <> [§ **6.5.4**]

interface_subprogram_specification ::= [§ **6.5.4**]
    interface_procedure_specification | interface_function_specification

interface_type_declaration ::= [§ **6.5.3**]
    interface_incomplete_type_declaration

interface_variable_declaration ::= [§ **6.5.2**]
    [**variable**] identifier_list : [mode] subtype_indication [:= *static*_expression]

iteration_scheme ::= [§ **10.10**]
      while condition
    | **for** *loop*_parameter_specification

label ::= identifier [§ **11.8**]

letter ::= upper_case_letter | lower_case_letter [§ **15.4.2**]

letter_or_digit ::= letter | digit [§ **15.4.2**]

library_clause ::= **library** logical_name_list ; [§ **13.2**]

library_unit ::= [§ **13.1**]
      primary_unit
    | secondary_unit

literal ::= [§ **9.3.2**]
      numeric_literal
    | enumeration_literal
    | string_literal
    | bit_string_literal
    | **null**

logical_expression ::= [§ **9.1**]
      relation { **and** relation }
    | relation { **or** relation }
    | relation { **xor** relation }
    | relation [**nand** relation]
    | relation [**nor** relation]
    | relation { **xnor** relation }

logical_name ::= identifier [§ **13.2**]

logical_name_list ::= logical_name { , logical_name } [§ **13.2**]

logical_operator ::= **and** | **or** | **nand** | **nor** | **xor** | **xnor** [§ **9.2.1**]

loop_statement ::= [§ **10.10**]
    [*loop*_label :]
        [iteration_scheme] **loop**
            sequence_of_statements
        end loop [*loop*_label] ;

miscellaneous_operator ::= ** | **abs** | **not** [§ **9.2.1**]

mode ::= **in** | **out** | **inout** | **buffer** | **linkage** [§ **6.5.2**]

multiplying_operator ::= ***** | / | **mod** | **rem** [§ **9.2.1**]

name ::= [§ **8.1**]
simple_name
| operator_symbol
| character_literal
| selected_name
| indexed_name
| slice_name
| attribute_name
| external_name

next_statement ::= [§ **10.11**]
[label :] **next** [*loop*_label] [**when** condition] ;

null_statement ::= [label :] **null** ; [§ **10.14**]

numeric_literal ::= [§ **9.3.2**]
abstract_literal
| physical_literal

object_declaration ::= [§ **6.4.2.1**]
constant_declaration
| signal_declaration
| variable_declaration
| file_declaration

operator_symbol ::= string_literal [§ **4.2.1**]

package_body ::= [§ **4.8**]
package body *package*_simple_name **is**
package_body_declarative_part
end [**package body**] [*package*_simple_name] ;

package_body_declarative_item ::= [§ **4.8**]
subprogram_declaration
| subprogram_body
| subprogram_instantiation_declaration
| package_declaration
| package_body
| package_instantiation_declaration
| type_declaration
| subtype_declaration
| constant_declaration
| variable_declaration
| file_declaration
| alias_declaration
| attribute_declaration
| attribute_specification
| use_clause
| group_template_declaration
| group_declaration

```
package_body_declarative_part ::=                                    [§ 4.8]
    { package_body_declarative_item }

package_declaration ::=                                              [§ 4.7]
    package identifier is
        package_header
        package_declarative_part
    end [ package ] [ package_simple_name ] ;

package_declarative_item ::=                                         [§ 4.7]
      subprogram_declaration
    | subprogram_instantiation_declaration
    | package_declaration
    | package_instantiation_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | component_declaration
    | attribute_declaration
    | attribute_specification
    | disconnection_specification
    | use_clause
    | group_template_declaration
    | group_declaration
    | PSL_Property_Declaration
    | PSL_Sequence_Declaration

package_declarative_part ::=                                         [§ 4.7]
    { package_declarative_item }

package_header ::=                                                   [§ 4.7]
    [ generic_clause
    [ generic_map_aspect ; ] ]

package_instantiation_declaration ::=                                [§ 4.9]
    package identifier is new uninstantiated_package_name
        [ generic_map_aspect ] ;

package_pathname ::=
    @ library_logical_name . { package_simple_name . } object_simple_name   [§ 8.7]

parameter_specification ::=                                          [§ 10.10]
    identifier in discrete_range

partial_pathname ::= { pathname_element . } object_simple_name       [§ 8.7]

pathname_element ::=                                                 [§ 8.7]
      entity_simple_name
    | component_instantiation_label
```

```
    | block_label
    | generate_statement_label [ ( static_expression ) ]
    | package_simple_name

physical_literal ::= [ abstract_literal ] unit_name                    [§ 5.2.4.1]

physical_type_definition ::=                                           [§ 5.2.4.1]
    range_constraint
        units
            primary_unit_declaration
            { secondary_unit_declaration }
        end units [ physical_type_simple_name ]

port_clause ::=                                                        [§ 6.5.6.3]
    port ( port_list ) ;

port_list ::= port_interface_list                                      [§ 6.5.6.3]

port_map_aspect ::=                                                    [§ 6.5.7.3]
    port map ( port_association_list )

prefix ::=                                                             [§ 8.1]
      name
    | function_call

primary ::=                                                            [§ 9.1]
      name
    | literal
    | aggregate
    | function_call
    | qualified_expression
    | type_conversion
    | allocator
    | ( expression )

primary_unit ::=                                                       [§ 13.1]
      entity_declaration
    | configuration_declaration
    | package_declaration
    | package_instantiation_declaration
    | context_declaration
    | PSL_Verification_Unit

primary_unit_declaration ::= identifier ;                              [§ 5.2.4.1]

procedure_call ::= procedure_name [ ( actual_parameter_part ) ]        [§ 10.7]

procedure_call_statement ::= [ label : ] procedure_call ;              [§ 10.7]

procedure_specification ::=                                            [§ 4.2.1]
    procedure designator
        subprogram_header
        [ [ parameter ] ( formal_parameter_list ) ]
```

process_declarative_item ::= [§ **11.3**]
    subprogram_declaration
  | subprogram_body
  | subprogram_instantiation_declaration
  | package_declaration
  | package_body
  | package instantiation_declaration
  | type_declaration
  | subtype_declaration
  | constant_declaration
  | variable_declaration
  | file_declaration
  | alias_declaration
  | attribute_declaration
  | attribute_specification
  | use_clause
  | group_template_declaration
  | group_declaration

process_declarative_part ::= [§ **11.3**]
  { process_declarative_item }

process_sensitivity_list ::= **all** | sensitivity_list [§ **11.3**]

process_statement ::= [§ **11.3**]
  [*process*_label :]
    [**postponed**] **process** [(process_sensitivity_list)] [**is**]
      process_declarative_part
    begin
      process_statement_part
    end [**postponed**] **process** [*process*_label] ;

process_statement_part ::= [§ **11.3**]
  { sequential_statement }

protected_type_body ::= [§ **5.6.3**]
  protected body
    protected_type_body_declarative_part
  end protected body [*protected_type*_simple name]

protected_type_body_declarative_item ::= [§ **5.6.3**]
    subprogram_declaration
  | subprogram_body
  | subprogram_instantiation_declaration
  | package_declaration
  | package_body
  | package_instantiation_declaration
  | type_declaration
  | subtype_declaration
  | constant_declaration
  | variable_declaration
  | file_declaration
  | alias_declaration
  | attribute_declaration

```
    | attribute_specification
    | use_clause
    | group_template_declaration
    | group_declaration

protected_type_body_declarative_part ::=                                    [§ 5.6.3]
    { protected_type_body_declarative_item }

protected_type_declaration ::=                                              [§ 5.6.2]
    protected
        protected_type_declarative_part
    end protected [ protected_type_simple_name ]

protected_type_declarative_item ::=                                         [§ 5.6.2]
     subprogram_declaration
    | subprogram_instantiation_declaration
    | attribute_specification
    | use_clause

protected_type_declarative_part ::=                                         [§ 5.6.2]
    { protected_type_declarative_item }

protected_type_definition ::=                                               [§ 5.6.1]
     protected_type_declaration
    | protected_type_body

qualified_expression ::=                                                    [§ 9.3.5]
     type_mark ' ( expression )
    | type_mark ' aggregate

range ::=                                                                   [§ 5.2.1]
     range_attribute_name
    | simple_expression direction simple_expression

range_constraint ::= range range                                            [§ 5.2.1]

record_constraint ::=                                                       [§ 5.3.3]
    ( record_element_constraint { , record_element_constraint } )

record_element_constraint ::= record_element_simple_name element_constraint  [§ 5.3.3]

record_element_resolution ::= record_element_simple_name resolution_indication  [§ 6.3]

record_resolution ::= record_element_resolution { , record_element_resolution }  [§ 6.3]

record_type_definition ::=                                                  [§ 5.3.3]
    record
        element_declaration
       { element_declaration }
    end record [ record_type_simple_name ]

relation ::=                                                                [§ 9.1]
    shift_expression [ relational_operator shift_expression ]
```

relational_operator ::= = | /= | < | <= | > | >= | ?= | ?/= | ?< | ?<= | ?> | ?>= [§ **9.2.1**]

relative_pathname ::= { ^ . } partial_pathname [§ **8.7**]

report_statement ::= [§ **10.4**]
    [label :]
        report expression
            [**severity** expression] ;

resolution_indication ::= [§ **6.3**]
    *resolution_function*_name | (element_resolution)

return_statement ::= [§ **10.13**]
    [label :] **return** [expression] ;

scalar_type_definition ::= [§ **5.2.1**]
      enumeration_type_definition
    | integer_type_definition
    | floating_type_definition
    | physical_type_definition

secondary_unit ::= [§ **13.1**]
      architecture_body
    | package_body

secondary_unit_declaration ::= identifier = physical_literal ; [§ **5.2.4.1**]

selected_expressions ::= [§ **10.5.4**]
    { expression **when** choices , }
     expression **when** choices

selected_force_assignment ::= [§ **10.5.4**]
    with expression **select** [?]
        target <= **force** [force_mode] selected_expressions ;

selected_name ::= prefix . suffix [§ **8.3**]

selected_signal_assignment ::= [§ **10.5.4**]
      selected_waveform_assignment
    | selected_force_assignment

selected_variable_assignment ::= [§ **10.6.4**]
    with expression **select** [?]
        target := selected_expressions ;

selected_waveform_assignment ::= [§ **10.5.4**]
    with expression **select** [?]
        target <= [delay_mechanism] selected_waveforms ;

selected_waveforms ::= [§ **10.5.4**]
    { waveform **when** choices , }
    waveform **when** choices

sensitivity_clause ::= **on** sensitivity_list [§ **10.2**]

sensitivity_list ::= *signal*_name { , *signal*_name } [§ **10.2**]

sequence_of_statements ::= [§ **10.1**]
    { sequential_statement }

sequential_statement ::= [§ **10.1**]
      wait_statement
    | assertion_statement
    | report_statement
    | signal_assignment_statement
    | variable_assignment_statement
    | procedure_call_statement
    | if_statement
    | case_statement
    | loop_statement
    | next_statement
    | exit_statement
    | return_statement
    | null_statement

shift_expression ::= [§ **9.1**]
    simple_expression [shift_operator simple_expression]

shift_operator ::= **sll** | **srl** | **sla** | **sra** | **rol** | **ror** [§ **9.2.1**]

sign ::= + | – [§ **9.2.1**]

signal_assignment_statement ::= [§ **10.5.1**]
      [label :] simple_signal_assignment
    | [label :] conditional_signal_assignment
    | [label :] selected_signal_assignment

signal_declaration ::= [§ **6.4.2.3**]
    signal identifier_list : subtype_indication [signal_kind] [:= expression] ;

signal_kind ::= **register** | **bus** [§ **6.4.2.3**]

signal_list ::= [§ **7.4**]
      *signal*_name { , *signal*_name }
    | **others**
    | **all**

signature ::= [[type_mark { , type_mark }] [**return** type_mark]] [§ **4.5.3**]

simple_configuration_specification ::= [§ **7.3.1**]
    for component_specification binding_indication ;
    [**end for** ;]

simple_expression ::= [§ **9.1**]
    [sign] term { adding_operator term }

simple_force_assignment ::= [§ **10.5.2.1**]
    target <= **force** [force_mode] expression ;

```
simple_name ::= identifier                                          [§ 8.2]

simple_release_assignment ::=                                       [§ 10.5.2.1]
    target <= release [ force_mode ] ;

simple_signal_assignment ::=                                        [§ 10.5.2.1]
      simple_waveform_assignment
    | simple_force_assignment
    | simple_release_assignment

simple_waveform_assignment ::=                                      [§ 10.5.2.1]
    target <= [ delay_mechanism ] waveform ;

simple_variable_assignment ::=                                      [§ 10.6.2.1]
    target := expression ;

slice_name ::= prefix ( discrete_range )                            [§ 8.5]

string_literal ::= " { graphic_character } "                        [§ 15.7]

subprogram_body ::=                                                 [§ 4.3]
    subprogram_specification is
        subprogram_declarative_part
    begin
        subprogram_statement_part
    end [ subprogram_kind ] [ designator ] ;

subprogram_declaration ::=                                          [§ 4.2.1]
    subprogram_specification ;

subprogram_declarative_item ::=                                     [§ 4.3]
      subprogram_declaration
    | subprogram_body
    | subprogram_instantiation_declaration
    | package_declaration
    | package_body
    | package_instantiation_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause
    | group_template_declaration
    | group_declaration

subprogram_declarative_part ::=                                     [§ 4.3]
    { subprogram_declarative_item }

subprogram_header ::=                                               [§ 4.2.1]
```

[**generic** (generic_list)
[generic_map_aspect]]

subprogram_instantiation_declaration ::= [§ **4.4**]
subprogram_kind identifier **is new** *uninstantiated_subprogram*_name [signature]
[generic_map_aspect] ;

subprogram_kind ::= **procedure** | **function** [§ **4.3**]

subprogram_specification ::= [§ **4.2.1**]
procedure_specification | function_specification

subprogram_statement_part ::= [§ **4.3**]
{ sequential_statement }

subtype_declaration ::= [§ **6.3**]
subtype identifier is subtype_indication ;

subtype_indication ::= [§ **6.3**]
[resolution_indication] type_mark [constraint]

suffix ::= [§ **8.3**]
simple_name
| character_literal
| operator_symbol
| **all**

target ::= [§ **10.5.2.1**]
name
| aggregate

term ::= [§ **9.1**]
factor { multiplying_operator factor }

timeout_clause ::= **for** *time*_expression [§ **10.2**]

tool_directive ::= ` identifier { graphic_character } [§ **15.11**]

type_conversion ::= type_mark (expression) [§ **9.3.6**]

type_declaration ::= [§ **6.2**]
full_type_declaration
| incomplete_type_declaration

type_definition ::= [§ **6.2**]
scalar_type_definition
| composite_type_definition
| access_type_definition
| file_type_definition
| protected_type_definition

type_mark ::= [§ **6.3**]
*type*_name
| *subtype*_name

unbounded_array_definition ::= [§ **5.3.2.1**]
    array (index_subtype_definition { , index_subtype_definition })
        of *element*_subtype_indication

use_clause ::= [§ **12.4**]
    use selected_name { , selected_name } ;

variable_assignment_statement ::= [§ **10.6.1**]
      [label :] simple_variable_assignment
    | [label :] conditional_variable_assignment
    | [label :] selected_variable_assignment

variable_declaration ::= [§ **6.4.2.4**]
    [**shared**] **variable** identifier_list : subtype_indication [:= expression] ;

verification_unit_binding_indication ::= [§ **7.3.4**]
    use vunit verification_unit_list

verification_unit_list ::= *verification_unit*_name { , *verification_unit*_name } [§ **7.3.4**]

wait_statement ::= [§ **10.2**]
    [label :] **wait** [sensitivity_clause] [condition_clause] [timeout_clause] ;

waveform ::= [§ **10.5.2.1**]
      waveform_element { , waveform_element }
    | **unaffected**

waveform_element ::= [§ **10.5.2.2**]
      *value*_expression [**after** *time*_expression]
    | **null** [**after** *time*_expression]

Annex D

(informative)

Potentially nonportable constructs

This annex lists those VHDL constructs whose use may result in nonportable descriptions.

A description is considered portable if it

a) Compiles, elaborates, initializes, and simulates to termination of the simulation cycle on all conformant implementations, and

b) The time-variant state of all signals and variables in the description are the same at all times during the simulation,

under the condition that the same stimuli are applied at the same times to the description. The stimuli applied to a model include the values supplied to generics and ports at the root of the design hierarchy of the model, if any.

Note that the content of files generated by a description are not part of the state of the description, but that the content of files consumed by a description are part of the state of the description.

The use of the following constructs may lead to nonportable VHDL descriptions:

— Resolution functions that do not treat all inputs symmetrically

— The comparison of floating-point values

— Events on floating-point-valued signals

— The use of explicit type conversion to convert floating-point values to integer values

— Any value that does not fall within the minimum guaranteed range for the type

— The use of architectures and subprogram bodies implemented via the foreign language interface (the 'FOREIGN attribute)

— Processes that communicate via file I/O, including TEXTIO

— Impure functions

— Linkage ports

— Ports and generics in the root of a design hierarchy

— Use of a time resolution greater than 1 fs

— Shared variables

— Procedure calls passing a single object of an array or record type to multiple formals where at least one of the formals is of mode **out** or **inout**

— Models that depend on a particular format of T'IMAGE

— Declarations of integer or physical types that have a secondary unit whose position number is outside of the range –(2**31–1) **to** 2**31–1

— The predefined attributes 'INSTANCE_NAME or 'PATH_NAME, if the behavior of the model is dependent on the values returned by the attributes

— Use of a conversion specifier F, a, or A in the value for the FORMAT parameter of a call to the predefined function TO_STRING

Annex E

(informative)

Changes from IEEE Std 1076-2002

This annex lists those clauses that have been changed from IEEE Std 1076-2002, during its revision. The clause numbers are from this present revision. Note that purely editorial changes, such as typographic error corrections and changes made to conform to IEEE terminological rules, are not listed.

Clause 1

— 1.3.5: Describes incorporation of PSL.

Clause 3

— 3.2.2: Descriptions of generic (formerly 1.1.1.1) and ports (formerly 1.1.1.2) moved to 6.5.6.2 and 6.5.6.3, respectively.

— 3.2.3: Addition of subprogram instantiations, package declarations and instantiations, and PSL declarations in entity declarative part.

— 3.2.4: Addition of PSL directives in entity statement part.

— 3.3.1: Change to scope of architecture body.

— 3.3.2: Addition of subprogram instantiations, package declarations and instantiations, and PSL declarations in block declarative part.

— 3.4.1: Addition of verification unit binding indication in configuration declaration.

— 3.4.2: Rules for configuring generate statements extended to include if-generate and case-generate statements; addition of specification that discrete range in a generate specification be static.

— 3.4.3: Addition of verification unit binding indication in component configuration.

Clause 4

— 4.2.1: Addition of subprogram header; description of uninstantiated and generic-mapped subprograms.

— 4.2.2.2: Changes for VHPI; changes to subtype rules.

— 4.2.2.3: Changes to subtype rules.

— 4.3: Addition of subprogram instantiations, package declarations and instantiations, and PSL declarations in subprogram declarative part; additional rules relating to uninstantiated subprograms.

— 4.4: New subclause describing subprogram instantiation declarations.

— 4.5.2: Extension to operator overloading rules.

— 4.5.3: Extension to rule relating to appearance of a signature.

— 4.6: Revision relating to new array subtype rules; other clarifications and corrections.

— 4.7: Addition of package header; description of uninstantiated and generic-mapped packages; addition of subprogram instantiations, package declarations and instantiations, and PSL declarations in package declarative part; rules relating to packages appearing other than as a design unit.

— 4.8: Addition of subprogram instantiations, package declarations and instantiations, PSL declarations, and attribute declarations and specifications in package body declarative part; rules relating to packages appearing other than as a design unit.

— 4.9: New subclause describing package instantiation declarations.

— 4.10: Addition of profile conformance, distinguished from lexical conformance.

Clause 5

- 5.1: Clarification of predefined operations; definition of unconstrained, partially constrained, and fully constrained composite subtypes.
- 5.2.1: Clarification of range constraints and range bounds of scalar types.
- 5.2.2: Clarification of enumeration literal declaration.
- 5.2.4.2: Change to definition of resolution limit.
- 5.2.6: New subclause describing predefined operations on scalar types.
- 5.3.2.1: Extensions to array type definitions; addition of array constraints, extending index constraints.
- 5.3.2.2: Extension and clarification of rules relating to index ranges and array subtypes.
- 5.3.2.3: Addition of new predefined array types.
- 5.3.2.4: New subclause describing predefined operations on array types.
- 5.3.3: Extensions to rules for record type definitions; addition of record constraints.
- 5.4.1: Clarification and revision of type rules.
- 5.5.1: Clarification and revision of type rules.
- 5.5.2: Addition of FLUSH operation; minor revisionof type rule.
- 5.6.2: Addition of subprogram instantiation in protected type declarative part; clarification of type rules.
- 5.6.3: Addition of subprogram instantiations, and package declarations and instantiations in protected type body declarative part.
- 5.7: New subclause describing string representations, formerly in 16.4.

Clause 6

- 6.1: Addition of subprogram instantiations, package instantiations, PSL verification units, and PSL declarations in declarations; clarification of the definition of a declaration; added definition of a designator.
- 6.2: Additional rules for generic types and type declarations.
- 6.3: Extension of subtype indication relating to constraints and resolution functions.
- 6.4.1: Change in terminology for generics.
- 6.4.2.2: Clarifications.
- 6.4.2.3: Revision of rules relating to resolved signals; clarification of type rules.
- 6.4.2.4: Revision of rules relating to appearance of variable declarations.
- 6.5: Former 4.3.2 promoted to this subclause.
- 6.5.1: Revision of definition of interface declaration.
- 6.5.2: Descriptions of interface objects demoted into this new subclause; revisions to rules relating to interface constants; revision of rules relating to modes.
- 6.5.3: New subclause describing interface type declarations.
- 6.5.4: New subclause describing interface subprogram declarations.
- 6.5.5: New subclause describing interface package declarations.
- 6.5.6.1: Revisions relating to new classes of interface declarations and new places where interface lists may appear.
- 6.5.6.2: New subclause containing descriptions from former 1.1.1 and 1.1.1.1; extensions relating to generic types, subprograms, and packages.
- 6.5.6.3: New subclause containing descriptions from former 1.1.1 and 1.1.1.2; extension relating to non-static expression associated with a port; extension relating to port modes.

— 6.5.7.1: Extensions relating to new classes of interface declarations and new places where association lists may appear; extensions to actual designators; clarifications of rules for type conversions and conversion functions.
— 6.5.7.2: New subclause containing descriptions from former 5.2.1.2; extensions relating to new classes of generics and new places where generic map aspects may appear.
— 6.5.7.3: New subclause containing descriptions from former 5.2.1.2.
— 6.6: Former 4.3.3 promoted to this subclause.
— 6.6.2: Extension to type rules; addition of rules relating to external names.
— 6.6.3: Clarification of rules relating to types.
— 6.7: Clarification of type rules.
— 6.11: New subclause describing PSL clock declarations.

Clause 7

— 7.2: Additional entity classes defined; revision of rules relating to specification of named entities; revision of rules relating to decoration of packages.
— 7.3.1: Addition of verification unit binding indication in configuration specification.
— 7.3.2: Extensions relating to new classes of interface declarations; clarification of error condition; descriptions in former 5.2.1.2 moved to 6.5.7.
— 7.3.3: Clarification.
— 7.3.4: New subclause describing verification unit binding indications.
— 7.4: Clarification of type rules.

Clause 8

— 8.1: Addition of character literals and external names; deleted rule relating to access-typed prefix.
— 8.3: Revision of expanded name rules relating to architecture bodies; addition of protected type definition in expanded name; clarification of prefix for a method name.
— 8.6: Clarification of predefined attribute names.
— 8.7: New subclause describing external names.

Clause 9

— 9.1: Addition of operators.
— 9.2.1: Addition of operators and predefined operations.
— 9.2.2: Extensions relating to operand types and unary forms.
— 9.2.3: Addition of matching relational operators.
— 9.2.4: Clarification of rule for index ranges.
— 9.2.5: Correction of terminology relating to index ranges.
— 9.2.7: Extension relating to operand types.
— 9.2.9: New subclause describing condition operator.
— 9.3.2: Revision of rules relating to null string literals; correction of terminology relating to index ranges.
— 9.3.3.1: Clarification of rules relating to discrete range choices.
— 9.3.3.3: Extensions relating to expressions of the type of the aggregate; revision and correction of terminology in rules relating to index ranges.
— 9.3.4: Additional rule relating to uninstantiated subprograms; clarification of parameter index range rules.
— 9.3.5: Revision of rule describing effect of type qualification.

— 9.3.6: Extension allowing conversion between array types with different index types; addition of rules relating to null arrays; extension relating to array types with undefined index ranges.

— 9.3.7: Extension to rules relating to constraints.

— 9.4.2: Addition of operations defined in STD_LOGIC_1164; revision and extension to rules describing locally static primaries, constraints, and subtypes.

— 9.4.3: Revision and extension to rules describing globally static primaries, constraints and subtypes.

Clause 10

— 10.2: Generalization of type of a condition; extended applicability of rules for forming sensitivity list.

— 10.3: Revision of rules relating to messages and severity level.

— 10.4: Revision of rules relating to messages and severity level.

— 10.5: Addition of force and release assignment; addition of simple, conditional and selected signal assignment.

— 10.5.2: New subclause containing descriptions from former 8.4, describing simple signal assignment, force and release.

— 10.5.2.2: Former 8.4.1; clarification of subtype rules; addition of rules relating to force and release.

— 10.5.3: New subclause containing descriptions from former 9.5.1 describing conditional signal assignment, force and release.

— 10.5.4: New subclause containing descriptions from former 9.5.2 describing selected signal assignment, force and release, and matching selected signal assignment.

— 10.6: Addition of simple, conditional, and selected variable assignment.

— 10.6.2: New subclause containing descriptions from former 8.5, describing simple variable assignment; clarification of subtype rules; extension dealing with VHPI force; deleted text originally intended to be non-normative; addition of rule relating to aggregate targets.

— 10.6.2.2: Former 8.5.1; revision of rules relating to composite target.

— 10.6.3: New subclause describing conditional variable assignment.

— 10.6.4: New subclause describing selected variable assignment.

— 10.7: Extension relating to uninstantiated subprograms; revision of rules relating to index ranges of parameters.

— 10.9: Addition of matching case statement; revision of rules relating to choices and expression subtype.

— 10.13: Revision to rules relating to result subtype.

Clause 11

— 11.1: Addition of PSL directive as a concurrent statement.

— 11.3: Addition of **all** in a process sensitivity list; addition of subprogram instantiations, and package declarations and instantiations in process declarative part.

— 11.5: Addition of rule disambiguating assertions.

— 11.6: Revision of concurrent signal assignment rules; former 9.5.1 moved to 10.5.3; former 9.5.2 moved to 10.5.4.

— 11.7.2: Revision of rules for equivalent block statements.

— 11.7.3: Revision of rules for equivalent block statements.

— 11.8: Addition of if-generate and case-generate statements.

Clause 12

- 12.1: Revision of rules relating to declaration of architecture bodies.
- 12.2: Revision of rules relating to scope of architecture names; additional rules relating to uninstantiated declarations and PSL verification units.
- 12.3: Revision of visibility rule determining more than one meaning; additional rules relating to packages, architecture names, subprogram and package instantiation, alternative labels of generate statements, PSL verification units, PSL declarations and PSL directives; revision of rules for visibility of interface objects; revision of visibility by selection rules; revision of homograph definition and hiding rules.
- 12.4: Revision of rules relating identified declarations; revision of rules relating to direct visibility; additional rule relating to context declarations.
- 12.5: Additional rule relating to subprogram instantiation.

Clause 13

- 13.1: Addition of package instantiation declaration, context declaration and PSL verification unit as primary units; additional rule relating to context declarations.
- 13.2: Additional rules relating to context declarations; addition of library IEEE and new packages.
- 13.3: New subclause describing context declarations.
- 13.4: Addition of context references.

Clause 14

- 14.2: Additional rules relating to VHPI and PSL verification units; extension to rule relating to top-level generics; additional rules relating to package instantiations; revision to order of elaboration of drivers; additional rules relating to external names.
- 14.3.1: Additional rules relating to package and subprogram headers.
- 14.3.2: Revision of order of elaboration.
- 14.3.3: Revision of order of elaboration; additional rules for new classes of generics.
- 14.3.4: Revision of order of elaboration.
- 14.3.5: Revision of order of elaboration; additional rules relating to association with non-static expressions; clarification of subtype rules.
- 14.4.1: Additional rules relating to VHPI.
- 14.4.2: Additional rule relating to PSL declarations.
- 14.4.2.2: Additional rules relating to subprogram instantiations.
- 14.4.2.3: Revision of rules relating to composite type declarations.
- 14.4.2.4: Additional rules relating to constraints.
- 14.4.2.5: Revision of rules relating to composite object declarations; clarified applicability of rules.
- 14.4.2.6: Revision of rule relating to composite alias declarations.
- 14.4.2.9: New subclause describing elaboration of packages.
- 14.4.3.2: Revision of rules relating to composite attributes.
- 14.5.1: Additional rules relating to VHPI and to PSL directives.
- 14.5.3: Additional rules relating to if-generate and case-generate statements.
- 14.5.5: Revision to order of elaboration of drivers.
- 14.6: Revision of order of elaboration of parameters; additional rules relating to VHPI.
- 14.7.1: Additional rule relating to PSL directives; revision of rules relating to kernel variables.
- 14.7.2: Revision and extension of rules relating to kernel variables, to update of drivers, and to VHPI.

— 14.7.3: Revision and extension of rules relating to force and release, update of drivers and signals, kernel variables, and VHPI.

— 14.7.4: Additional rules relating to VHPI.

— 14.7.5: Former 12.6.4 partitioned into subclauses.

— 14.7.5.1: New subclause containing description of initialization from former 12.6.4; revised rules for determining T_n factored out.

— 14.7.5.2: New subclause containing description of initialization from former 12.6.4; additional rules for VHPI and for force and release; revision of rules relating to order of signal update; additional rules relating to PSL directives.

— 14.7.5.3: New subclause containing description of the simulation cycle from former 12.6.4; additional rules for VHPI and for force and release; revision of rules relating to order of signal update; additional rules relating to PSL directives.

Clause 15

— Clause 15: Former 13.10 deleted.

— 15.2: Additional rules relating to tool directives and to PSL; revision of character set usage; clarification of case correspondence.

— 15.3: Additional rules relating to tool directives and to PSL; additional delimiters.

— 15.8: Addition of length, signed and unsigned bases, decimal base, and non-digit characters.

— 15.9: Addition of delimited comments.

— 15.10: Additional reserved words defined; additional rules for PSL.

— 15.11: New subclause describing tool directives .

Clause 16

— 16.2: Additional rule relating to predefined attributes that are functions; clarification of atribute kinds; revision of rules for 'VALUE, 'IMAGE, 'POS, 'LENGTH, 'ASCENDING, and 'LAST_VALUE; revision of rules for attributes of arrays; addition of 'SUBTYPE and 'ELEMENT; additional rules for 'PATH_NAME and 'INSTANCE_NAME.

— 16.3: Additional types and operations defined.

— 16.4 Additional operations defined; description of string representation moved to 5.7.

— 16.5: New subclause describing standard environment package.

— 16.6: New subclause describing standard mathematical packages, formerly in IEEE 1076.2-1996 [B11].

— 16.7: New subclause describing standard multivalue logic package, formerly in IEEE 1164-1993 [B16].

— 16.8: New subclause describing standard synthesis packages, formerly in IEEE 1076.3-1997 [B12].

— 16.9: New subclause describing standard synthesis context declarations.

— 16.10: New subclause describing fixed-point packages.

— 16.11: New subclause describing floating-point packages.

Clauses 17–23

— New clauses describing VHPI.

Clause 24

— New clause describing standard tool directives.

Annex A

— New annex describing accompanying files.

Annex B

— New annex describing VHPI header files and extension mechanisms.

Annex C

— Formerly Annex A; revision of productions and addition of new productions.

Annex D

— Formerly Annex C; additional item relating to format specifiers.

Annex F

— Formerly Annex E; deletion of entries.

Annex G

— New annex describing use of standard packages.

Annex H

— New annex describing use of protect directive.

Annex I

— Formerly Annex B; revision of entries and addition of new entries.

Annex J

— Formerly Annex F; revision of entries and addition of new entries.

Annex F

(informative)

Features under consideration for removal

The following features are being considered for removal from a future version of the language.[17] Accordingly, modelers should refrain from using them when possible:

— None

[17]To comment on these, or any other features of VHDL, please visit http://www.eda.org/vasg.

Annex G

(informative)

Guide to use of standard packages

G.1 Using the MATH_REAL and MATH_COMPLEX packages

G.1.1 General

The information in this clause is intended to be a brief guide to using the MATH_REAL and MATH_COMPLEX packages, but it is not a normative part of the standard. As a standard, this set of packages provides a means of building models that interoperate and port to different tools, provided that the user adheres to a set of guidelines required by the standard and the strict typing imposed by the VHDL language.

G.1.2 Package bodies for MATH_REAL and MATH_COMPLEX

The collection of machine-readable files that forms part of this standard includes package bodies for MATH_REAL and MATH_COMPLEX. These package bodies are intended to provide a guideline for implementors. They are not a normative part of this standard, but suggest ways in which implementors may implement the MATH_REAL and MATH_COMPLEX packages. Implementors may also use the package bodies as a guideline to verify their implementation of the packages.

G.1.3 Predefined data types, operators, and precision for MATH_REAL

The MATH_REAL package is built on top of the standard data type (REAL), operators, and precision requirements for floating-point operations defined in STD.STANDARD.

G.1.4 Use and constraints of pseudo-random number generator in MATH_REAL

The pseudo-random number generator provided with the package is platform independent. In order to generate a chain of pseudo-random numbers, the seed values shall be set only in the first call to the function. A different chain of numbers is started every time the seed values are set. If multiple chains of pseudo-random numbers are required, then different sets of seed values have to be used for every chain.

G.1.5 Precision across different platforms

It is important to note that the math package results may be slightly different on different platforms because of variations in hardware support for floating-point arithmetic. These differences might not be immediately apparent to the average VHDL user. However, since most workstations use the IEEE 754 floating-point format, the variations are likely to be limited in practice.

G.1.6 Handling of overflow/underflow conditions

The detection of underflow/overflow is optional and implementation dependent.

G.1.7 Testbench for the packages

A non-exhaustive testbench for the packages MATH REAL and MATH COMPLEX can be found in the collection of machine-readable files that forms part of this standard.

G.1.8 Overloading side effect

Note that there is a side effect of adding functions for COMPLEX_POLAR when numerical expressions are used. Numerical parameters for these functions are ambiguous, unless a qualifier is used to disambiguate them. For example, SIN((0.0, 0.0)) is ambiguous. One has to say either SIN(COMPLEX'(0.0, 0.0)) or SIN(COMPLEX_POLAR'(0.0, 0.0)).

G.1.9 Synthesizability of functions

Synthesizability of the functions defined in the mathematical packages is beyond the scope of this standard.

G.2 Using the STD_LOGIC_1164 package

G.2.1 General

This subclause is intended to be a brief guide to using the STD_LOGIC_1164 package. This package provides a means of building models that interoperate, provided that the user adheres to a set of guidelines required by the strict typing imposed by the VHDL language.

G.2.2 Value system

The value system in STD_LOGIC_1164 was developed to model a variety of digital device technologies. The base type of the logic system is named “std_ulogic” where the “u” in the name signifies “unresolved.” Each of the elements comprising the type have a specified semantic and a commonly used application. In order for models to properly interoperate, one must interpret the meaning of each of the elements as provided by the standard.

Value	Name	Usage
'U'	Uninitialized state	Used as a default value
'X'	Forcing unknown	Bus contentions, error conditions, etc.
'0'	Forcing zero	Transistor driven to GND
'1'	Forcing one	Transistor driven to VCC
'Z'	High impedance	3-state buffer outputs
'W'	Weak unknown	Bus terminators
'L'	Weak zero	Pull down resistors
'H'	Weak one	Pull up resistors
'–'	Don’t care	Used for synthesis and advanced modeling

G.2.3 Handling strengths

Behavioral modeling techniques rarely require knowledge of the strength of a signal's value. Therefore, a number of "strength stripper" functions have been designed to transform 'Z', 'W', 'L', 'H', and '-' into their corresponding "forcing" strength counterparts.

Once in forcing strength, the model can simply respond to X's, 0's, l's, and 'U's as the need may arise. This strength stripping is done by using one of the following functions:

To_X01 (...)	converts 'L' and 'H' to '0' and '1' respectively. All others are converted to 'X'.
To_UX0 1 (...)	converts 'L' and 'H' to '0' and '1' respectively. 'U's are propagated and all others are converted to 'X'.

G.2.4 Use of the uninitialized value

The 'U' value is located in the first position of the type. Therefore, any object declared to be of this base type will be automatically initialized to 'U' unless expressly assigned a default expression.

Uninitialized values were designed to provide a means of detecting system values that have not changed from their uninitialized state since the time of system initialization. Hence, the logical tables for AND, OR, NAND, NOR, XOR, XNOR, and NOT have been designed to propagate 'U' states whenever encountered.

The propagation of 'U's through a circuit gives the designer an understanding of where the system has failed to be properly initialized.

G.2.5 Behavioral modeling for 'U' propagation

For behavioral modeling where 'U' propagation is desired, the function TO_UX01 will provide a reduction in the state system, as far as the modeler is concerned, thereby easing the modeler's task.

G.2.6 'U's related to conditional expressions

Case statements, "if" expressions, and selected signal assignments need to separately treat 'U' states and provide a path for 'U' state propagation in order to propagate 'U's.

G.2.7 Structural modeling with logical tables

The logical tables are designed to generate output values in the range 'U', 'X', '0', and '1'. Therefore, once an element of the nine-state system passes through any of the logical tables, it will be converted to forcing strength. If the need arises for a weak or floating strength to be propagated through the remainder of a circuit or to an output port, then the model developer shall be certain to assign the appropriate value accordingly.

G.2.8 X-handling: assignment of X's

In assignments, the 'X' and '–' values differ minimally. The value '–', also known as "output don't care," explicitly means that synthesis tools are allowed to generate either a '0' or a '1', whichever leads to minimal circuitry, whereas 'X' usually appears during transitions or as a result of bus contentions or to flag model generated internal error conditions, such as in the following waveform assignment:

```
S <= 'X' after 1 ns, '1' after 5 ns;
```

where the current value of S becomes indeterminate after 1 ns and then reaches '1' after 5 ns have elapsed.

G.2.9 Modeling with don't care's

G.2.9.1 Use of the don't care state in synthesis models

For synthesis, a VHDL program is a specification of the functionality of a design. VHDL can also be used to model (in order to simulate) real circuits. The former deals with logical function of the circuit, while the latter is concerned with function of a circuit from an electrical point of view. The nine-state logic type usage for synthesis is based on the assumption that the VHDL models will be logical function specifications and, therefore, attempt to restrict the usage of the logic type to logical function. The motivation for allowing the user to reference the values 'U' and 'X' (which do not specify the behavior of the circuit to be built, i.e., one can not build a circuit which "drives an 'X'") is to allow such simulation artifacts to remain in models for synthesis for the sake of convenience. By having synthesis remove these references, the user is assuming only the kind of usage (of 'U' and 'X') that catches error states that should never occur in hardware.

G.2.9.2 Semantics of '-'

In designing the resolution function and the various logic tables in the package body, '–' is almost exclusively a syntactic shorthand for 'X', provided for compatibility with synthesis tools. This is evident from the fact that '–' becomes 'X' as soon as it is operated upon and when it is converted to subtype X01 or UX01. The "output don't care" value represents either a '1' or a '0' as the output of combinatorial circuitry, with respect to state encoding in particular.

G.2.10 Resolution function

In digital logic design, there are a number of occasions in which driving outputs of more than one device are connected together. The most common of which is TRI-STATE ®[18] buses in which memory data ports are connected to each other and to controlling microprocessors. Another common case is one in which multiple drivers are parallel driving a heavily loaded signal path. In each of these cases, the VHDL language requires that the signals used to interconnect those devices be "resolved" signal types.

Focusing on resolution: when two signals' values are driving the same "wire," some resulting value will be observed on that wire. For example, if two parallel buffers both drive '1' onto a signal, then the signal will be '1'. If a TRI-STATE driver is in the high-impedance state 'Z' and another driver is in the forcing one '1' state, then the combination of those two signal values will result in a value of '1' appearing on the wire.

The resolution function built into STD_LOGIC_1164 operates on the principal that weak values dominate over high-impedance values and forcing values dominate over weak values.

G.2.11 Using STD_ULOGIC vs. STD_LOGIC

In deciding whether to use the resolved signal or unresolved signal type, a number of considerations need to be made:

a) Does the simulator run slower when using a resolved type than when using an unresolved type, or is the simulator optimized for the STD_LOGIC data types?

[18]TRI-STATE is a registered trademark of National Semiconductor Corporation. This information is given for the convenience of users of this standard and does not constitute an endorsement by the IEEE of these products. Equivalent products may be used if they can be shown to lead to the same results.

b) How many sources are there for a signal?

Each of these is considered, in order, as follows:

In the absence of other considerations, the choice between an unresolved and a resolved signal type should depend on whether the signal is intended to have only one source, or whether multiple sources are intended. In the former case, a scalar signal should be of type STD_ULOGIC, since inadvertent connection of multiple sources can be detected during analysis or elaboration. Similarly, a vector signal should be of type STD_ULOGIC_VECTOR, for the same reason. In the latter case a scalar signal should be of type STD_LOGIC, and a vector signal should be of type STD_LOGIC_VECTOR.

The same considerations apply to ports, regardless of the actual signals to which they are connected. An input port can be of either an unresolved or a resolved type, as the question of sources is not relevant. An output or bidirectional port with one internal source should be of type STD_ULOGIC or STD_ULOGIC_VECTOR. An output or bidirectional port with multiple internal sources should be of type STD_LOGIC or STD_LOGIC_VECTOR. The values contributed by the internal sources are resolved to determine the value driven by the port. Since STD_LOGIC is a subtype of STD_ULOGIC, ports and signals of these types can be interconnected freely. Similarly, since STD_LOGIC_VECTOR is a subtype of STD_ULOGIC_VECTOR, ports and signals of these vector types can be interconnected freely.

G.3 Notes on the synthesis package functions

G.3.1 General

This subclause provides notes on functions included in the NUMERIC_BIT, NUMERIC_STD, NUMERIC_BIT_UNSIGNED, and NUMERIC_STD_UNSIGNED packages. Except where otherwise indicated, notes applying to operations on type UNSIGNED in NUMERIC_BIT and NUMERIC_STD also apply to operations on BIT_VECTOR in NUMERIC_BIT_UNSIGNED and STD_ULOGIC_VECTOR in NUMERIC_STD_UNSIGNED.

The appearance of a code fragment in this subclause does not require a synthesis tool conforming to this standard to accept the construct represented by that fragment.

G.3.2 General considerations

G.3.2.1 Mixing SIGNED and UNSIGNED operands

The NUMERIC_BIT and NUMERIC_STD packages do not provide functions for mixing SIGNED and UNSIGNED operands. To do so would make it necessary to use qualified expressions to disambiguate commonly occurring forms. For example, with the declarations

```
variable S: SIGNED (3 downto 0);
variable U: UNSIGNED (4 downto 0);
```

if the arithmetic and relational functions allowed mixing of SIGNED and UNSIGNED operands, it would be necessary to rewrite the expressions

```
S >= "0000"
```

and

```
U + "1"
```

as

```
S >= SIGNED'("0000")
```

and

```
U + UNSIGNED'("1")
```

To apply a binary operation from the NUMERIC_BIT or NUMERIC_STD package to a combination of SIGNED and UNSIGNED operands, the user must explicitly convert one of the operands to the other type (see G.3.6.2).

G.3.2.2 Mixing vector and element operands

The packages do not declare functions that combine a vector with an operand that belongs to the element type of the vector, other than the + and – functions. For example, with the declarations

```
signal A, B, S: SIGNED(3 downto 0);
signal C: BIT;
```

a user shall not write

```
S <= A * B(3);
```

or

```
S <= A * C;
```

or

```
S <= A / '1';
```

For the first and third example, a user may write instead

```
S <= A * B(3 downto 3);
```

and

```
S <= A / "1";
```

For the second example, the user may concatenate C with a 0-length vector

```
S <= A * (C & "");
```

G.3.3 Arithmetic operator functions

G.3.3.1 Overflow of maximum negative value

When the SIGNED operand to an **abs** (function A.1) or unary – (function A.2) function has the maximum negative value for the number of elements that it has, the result is the maximum negative value of the same size. This means, for example, that


```
- SIGNED'("1000")
```

evaluates to

```
"1000"
```

Similarly, in functions A.22 and A.25, when the first operand to the / operator has the maximum negative value for the number of elements that it has, and when the second operand is either an INTEGER with the value –1 or a SIGNED operand with a value equivalent to –1, the result is the same as the first operand, rather than its complement:

```
SIGNED'("1000") / "11111"       evaluates to "1000"
SIGNED'("10000") / (-1)         evaluates to "10000"
```

To prevent overflow, a user may add an extra bit to the representation. For example, with the declarations

```
variable DIVIDEND: SIGNED (4 downto 0);
variable DIVISOR: INTEGER range -8 to 7;
variable QUOTIENT: SIGNED (5 downto 0);
```

one may write

```
QUOTIENT := (DIVIDEND(4) & DIVIDEND) / DIVISOR;
```

G.3.3.2 Lack of carry and borrow

When both operands of a binary arithmetic function + or – are either SIGNED or UNSIGNED, the function returns a value with the same number of elements (bits) as the larger of the two operands. If one operand is SIGNED or UNSIGNED and the other is INTEGER or NATURAL, the function returns a value with the same number of elements as the vector operand. Thus, these functions do not return an extra bit to represent a carry, borrow, or overflow value, nor do they generate a warning if a carry, borrow, or overflow occurs.

The choice not to generate a carry or borrow (and not to generate a warning) makes it easier to represent counter operations in the VHDL source code via assignments such as

```
A := A + 1;
```

or

```
B <= B - "1";
```

To obtain the appropriate carry, borrow, or overflow value, a user may add an extra bit to the vector operand. For example, with the declarations

```
signal U: UNSIGNED (4 downto 0);
signal S: SIGNED (5 downto 0);
signal SUM: UNSIGNED (5 downto 0);
signal DIFFERENCE: SIGNED (6 downto 0);
```

one may write

```
SUM <= ('0' & U) + 1;
DIFFERENCE <= (S(5) & S) - "1";
```

G.3.3.3 Return value for metalogical and high-impedance operands

If an operand to a NUMERIC_STD or NUMERIC_STD_UNSIGNED arithmetic function contains a metalogical or high-impedance value, the function returns a vector in which every element has the value 'X'. The function does not report a warning or error.

G.3.4 Relational operator functions

G.3.4.1 Justification of vector operands

The relational operator functions defined in the synthesis packages have a behavior different from the default behavior defined by this standard for vector types. The default behavior compares the vector elements left to right after the operands are left-justified, whereas the relational operator functions defined in the synthesis packages treat their operands as representing binary integers.

Table G.1 compares results for the predefined relational operators applied to BIT_VECTORs with the relational operators defined in the packages for SIGNED and UNSIGNED values. The results of relational operators defined in the NUMERIC_BIT_UNSIGNED package for BIT_VECTORs and in the NUMERIC_STD_UNSIGNED package for STD_ULOGIC_VECTORs are the same as the results for UNSIGNED.

Table G.1—Relational operators examples

Expression	Predefined operation on…	Package operation on…	
	BIT_VECTOR	UNSIGNED	SIGNED
"001" = "00001"	FALSE	TRUE	TRUE
"001" > "00001"	TRUE	FALSE	FALSE
"100" < "01000"	FALSE	TRUE	TRUE
"010" < "10000"	TRUE	TRUE	FALSE
"100" < "00100"	FALSE	FALSE	TRUE

G.3.4.2 Expansion of vector operands compared to integers

When a relational operator compares a SIGNED or UNSIGNED operand value with an INTEGER or NATURAL value, the function has the effect of converting the SIGNED or UNSIGNED operand to its equivalent universal integer value and then doing the corresponding comparison of integer values. For example

```
(SIGNED'("111") > -8) = TRUE
```

and

```
(UNSIGNED'("111") < 8) = TRUE
```

That is, the INTEGER value may be larger in magnitude than any value that can be represented by the number of elements in the SIGNED or UNSIGNED value.

G.3.4.3 Return value for metalogical and high-impedance operands

If an operand to any of the NUMERIC_STD or NUMERIC_STD_UNSIGNED relational operator functions for =, <, <=, >, or >= contains a metalogical or high-impedance value, the function returns the value FALSE. If an operand to the NUMERIC_STD or NUMERIC_STD_UNSIGNED relational operator function /= contains a metalogical or high-impedance value, the function returns the value TRUE.

G.3.5 Shift functions

G.3.5.1 Multiplication by a power of 2 with remaindering

The SHIFT_LEFT function for an UNSIGNED parameter provides for multiplication by a power of 2 remaindered by the maximum size of the vector parameter. In particular, if ARG is UNSIGNED and contains neither metalogical or high-impedance values, and if the integer values fall within the range allowed for INTEGERs:

```
TO_INTEGER (SHIFT_LEFT (ARG, COUNT)) =
   TO_INTEGER (ARG) * (2 ** COUNT) rem (2 ** ARG'LENGTH)
```

G.3.5.2 Division by a power of 2

The SHIFT_RIGHT function for an UNSIGNED parameter provides for division by a power of 2. That is, if ARG is UNSIGNED and contains neither metalogical or high-impedance values, and if the integer values fall with the range allowed for INTEGERs:

```
TO_INTEGER (SHIFT_RIGHT (ARG, COUNT) = TO_INTEGER (ARG) / (2 ** COUNT)
```

G.3.6 Type conversion functions

G.3.6.1 Overflow in conversion to INTEGER

The TO_INTEGER function does not contain code to check that the SIGNED or UNSIGNED parameter has an equivalent universal integer value that belongs to the range defined for the INTEGER or NATURAL subtypes. If TO_INTEGER is called with a parameter value that is too large, the simulation tool may therefore detect an overflow. A user should avoid applying TO_INTEGER to parameter subtypes for which the number of elements is greater than the number of bits used to represent INTEGERs in the user's simulation and synthesis tools.

G.3.6.2 Conversion between SIGNED and UNSIGNED

The packages do not provide functions for converting directly between the SIGNED and UNSIGNED types. Such conversions must be performed by the user. There are several ways to convert between SIGNED and UNSIGNED types. In performing such conversions, a user must determine how to handle any possible differences in the ranges supported by SIGNED and UNSIGNED objects having the same number of elements. For example, suppose the VHDL source code contains the declarations

```
signal S: SIGNED(3 downto 0);
signal BIG_S: SIGNED(4 downto 0);
signal U: UNSIGNED(3 downto 0);
constant S1: SIGNED(3 downto 0) := "1000"; -- equivalent to -8
constant U1: UNSIGNED(3 downto 0) := "1100"; -- equivalent to +12
```

a) A user can use a VHDL type conversion to convert one form to another:

```
S <= SIGNED (U1);   -- U1 (= +12) gets converted to S (= -4)
U <= UNSIGNED (S1); -- S1 (= -8)  gets converted to U (= +8)
```

b) A user can add an extra bit to represent the sign when converting from UNSIGNED to SIGNED:

```
BIG_S <= SIGNED ('0' & U1); -- U1 (= +12) gets converted
                            -- to BIG_S (= +12)
```

c) Finally, a user can generate an error or warning when the value of one cannot be represented in the number of elements available in the other:

```
assert S >= "0000"
   report "Cannot convert negative value."
   severity WARNING;
U <= UNSIGNED (S);
```

G.3.7 Logical operator functions

G.3.7.1 Application to SIGNED and UNSIGNED

The functions that define the application of the logical operators **and**, **or**, **nand**, **nor**, **xor**, and **xnor** to SIGNED and UNSIGNED operand values are equivalent to functions that apply the same logical operators to STD_LOGIC_VECTOR (or STD_ULOGIC_VECTOR) parameters. This equivalence includes the handling of metalogical and high-impedance element values. That is, for example, if S1 and S2 are SIGNED values of equal length:

```
S1 nand S2 = SIGNED (STD_LOGIC_VECTOR (S1) nand STD_LOGIC_VECTOR (S2))
```

G.3.7.2 Index range of return values

For the functions **and**, **or**, **nand**, **nor**, **xor**, and **xnor** defined in the NUMERIC_STD package, the index range for the return values has the form "*n* – 1 **downto** 0," where *n* is the number of elements in the return value.

In the NUMERIC_BIT package, the corresponding functions are defined implicitly by the type declarations for the SIGNED and UNSIGNED types, so that the index range of the return values is as defined by this standard (see 9.2.2).

G.3.8 The STD_MATCH function

The behavior of the STD_MATCH functions in the NUMERIC_STD package differs from that of the = functions for the same types of parameters. The STD MATCH function compares its parameters element by element, and treats the value '–' as matching any other STD_ULOGIC value. The = function interprets its operands, however, as representing the equivalent integer values, and returns TRUE if the equivalent integer values are equal.

G.4 Using the fixed-point package

G.4.1 General

Fixed point is a step between integer math and floating point. This has the advantage of being almost as fast as NUMERIC_STD arithmetic, but able to represent numbers that are less than 1.0. A fixed-point number has an assigned width and an assigned location for the binary point. As long as the number is big enough to

provide enough precision, fixed point is fine for most digital signal processing (DSP) applications. Because it is based on integer math, it is extremely efficient, as long as the data does not vary too much in magnitude.

The fixed-point package defines two types: “unresolved_ufixed” is the unsigned fixed point, and “unresolved_sfixed” is the signed fixed point.

```
type unresolved_ufixed is array (INTEGER range <>) of STD_ULOGIC;
type unresolved_sfixed is array (INTEGER range <>) of STD_ULOGIC;
```

There are also aliases of these types, “U_ufixed” and “U_sfixed”. The package defines subtypes, “ufixed” and “sfixed”, with resolved elements:

```
subtype ufixed is (resolved) unresolved_ufixed;
subtype sfixed is (resolved) unresolved_sfixed;
```

Example:

```
   use ieee.fixed_pkg.all;
   ...
   signal a, b: sfixed (7 downto -6);
   signal c: sfixed (8 downto -6);
begin
   ...
   c <= a + b;
```

The fixed-point data types define the location of the binary point by using negative indices within a descending index range. The binary point is assumed to be between the 0 and –1 index. Thus, given a declaration

```
signal y: ufixed (4 downto -5)
```

the data type represents unsigned fixed point, 10 bits wide, with 5 bits after the binary point. Then assigning y = 6.5 in decimal, or = 00110.10000 in binary, can be written:

```
y <= "0011010000";
```

The signed data type uses 2s-complement representation, just like the NUMERIC_STD package.

Any non-null index range is valid. Thus:

```
signal z: ufixed (-2 downto -3);
signal y: sfixed (3 downto 1);
...
z <= "11";  -- 0.011 = 0.375
y <= "111"; -- 1110.0 = -2
```

G.4.2 Literals and type conversions

Conversion functions have been created for INTEGER, REAL, SIGNED, and UNSIGNED types. These conversion functions can be called with two different sets of parameters, one set giving the index bounds of the result directly, and the other consisting of a single parameter whose index bounds are used. For example, to convert from a real number to a signed fixed-point result:

```
a <= to_sfixed (-3.125, 7, -6);
b <= to_sfixed (inp1, b); -- returns "inp1" sized the same as "b"
```

Likewise, to convert from a real number to an unsigned fixed-point result:

```
y <= to_ufixed (6.5, 4, -5);
```

where 4 is the upper index, and –5 is the lower index; or similarly:

```
y <= to_ufixed (6.5, y'high, y'low);
```

or:

```
y <= to_ufixed (6.5, y);
```

The to_signed and to_unsigned conversion functions are also overloaded to take the two forms of parameters specifying the result bounds. Rounding and saturation rules apply on these functions.

G.4.3 Sizing rules

The data widths in the fixed-point package are designed so that there is no possibility of an overflow. This is a departure from the NUMERIC_STD model, which simply throws away underflow and overflow bits. The index range of the result of an operation is defined in Table G.2.

Table G.2—Index range of result of an operation

Operation	Result range
A + B	Max(A'left, B'left) + 1 **downto** Min(A'right, B'right)
A – B	Max(A'left, B'left) + 1 **downto** Min(A'right, B'right)
A * B	A'left + B'left + 1 **downto** A'right + B'right
A **rem** B	Min(A'left, B'left) **downto** Min(A'right, B'right)
Signed A/B	A'left – B'right + 1 **downto** A'right – B'left
Signed A **mod** B	Min(A'left, B'left) **downto** Min(A'right, B'right)
Signed reciprocal(A)	–A'right **downto** –A'left – 1
abs A	A'left + 1 **downto** A'right
– A	A'left + 1 **downto** A'right
Unsigned A/B	A'left – B'right **downto** A'right – B'left – 1
Unsigned A **mod** B	B'left **downto** Min(A'right, B'right)
Unsigned reciprocal(A)	–A'right + 1 **downto** –A'left

Example:

Given the unsigned declarations:


```
signal x: ufixed (7 downto -3);
signal y: ufixed (2 downto -9);
```

Multiplying x by y gives a result of type ufixed (7+2+1 **downto** –3+(–9)), or ufixed (10 **downto** –12).

Given the signed declarations:

```
signal x: sfixed (-1 downto -3);
signal y: sfixed (3 downto 1);
```

Dividing x by y gives a result of type sfixed (–1–1+1 **downto** –3–3), or sfixed (–1 **downto** –6).

It is not necessary to memorize the size rules. Instead, the resize function can be used, or the functions ufixed_high, ufixed_low, sfixed_high, and sfixed_low can be used to return the bounds of an operand.

Example:

```
   variable a: sfixed (5 downto -3);
   variable b: sfixed (7 downto -9);
   variable adivb: sfixed (sfixed_high (5, -3, '/', 7, -9)
                           downto sfixed_low (5, -3, '/', 7, -9));
begin
   adivb <= a / b; -- signed fixed-point divide
```

Alternatively:

```
signal adivb:
         sfixed (sfixed_high (a'high, a'low, '/', b'high, b'low)
                 downto sfixed_low (a'high, a'low, '/', b'high, b'low));
```

or:

```
signal adivb: sfixed(sfixed_high (a, '/', b)
                     downto sfixed_low (a, '/', b));
```

The resize function can be used to fix the size of the output. However, rounding and saturate rules are applied:

```
x <= resize (x * y, x'high, x'low);
```

The increase in result size can cause problems in some designs, such as an accumulator, that is, a fixed-width number to which other numbers are added repeatedly. To implement an accumulator in the fixed-point packages, the resize function can be applied to the result of the addition, or the add_carry procedure can be used, as follows:

```
signal ACC: ufixed (7 downto -3);
...
add_carry ( L => ACC, R => X, C_in => '0', Result => ACC, C_out => open);
```

The divide function is defined as follows:

```
function divide (l, r: sfixed;
                 round_style: BOOLEAN := fixed_round_style;
                 guard_bits: NATURAL := fixed_guard_bits) return sfixed;
```

The output is sized with the same rules as the / operator. The function allows the number of guard bits and the rounding operation to be overridden. Note that the output size is calculated so that overflow is not possible.

The reciprocal function is defined in a similar manner to the divide function:

```
function reciprocal (arg: ufixed;
                     round_style: BOOLEAN := fixed_round_style;
                     guard_bits: NATURAL := fixed_guard_bits)
                     return ufixed;
```

This function performs the operation “1/arg”, with the output vector following the sizing rules as previously noted. The function is very useful for dividing by a constant. For example:

```
A := B / Cons;
```

can be rewritten as:

```
A := B * reciprocal (Cons);
```

because a multiplier typically uses less logic than a divider, this change can save significant hardware resources.

G.4.4 Rounding and saturation

Many of the fixed-point operations include parameters to control rounding and saturation behavior. An example is the resize operation, which may be called as follows:

```
X <= resize (arg => X + 1,
             left_index => X'high, right_index => X'low,
             overflow_style => fixed_wrap,
             round_style => fixed_truncate );
```

In the FIXED_PKG package, round_style defaults to fixed_round, which turns on the rounding routines. If round_style is fixed_truncate, the number is truncated. Rounding returns the representable value that is nearest the original value before dropping the remainder. If the remainder places the original value exactly in the middle of two representable values, the one with its least significant bit 0 is returned. The rounding operation is implemented by examining the least significant bit of the unrounded value and the bits of the remainder. If the most significant bit of the remainder is 1, and either the least significant bit of the unrounded value is 1 or any bits other than the most significant of the remainder (or both), then the unrounded value is rounded up; otherwise it is returned as is. While this has the advantage of maintaining accuracy, like floating-point round-nearest behavior, it has the disadvantage that all of the bits of the remainder must be examined to do rounding, increasing the hardware complexity.

In the FIXED_PKG package, overflow_style defaults to fixed_saturate: if the true result is too large to represent, the returned result is the maximum possible number. The alternative for overflow_style is fixed_wrap, where the top bits are simply truncated. Unlike in NUMERIC_STD, the sign bit is not preserved when wrapping. Thus, it is possible to get a positive result when resizing a negative number in this mode.

Finally, a guard_bits parameter on many operations defaults to the value of fixed_guard_bits, which is 3 in FIXED_PKG. Guard bits are used in the rounding routines. If guard_bits is 0, rounding is turned off.

Otherwise, the extra bits are added to the end of the numbers in the division and to_real functions to make the numbers more accurate.

G.4.5 Overloading

The following operations are defined for ufixed:

+, –, *, /, **rem**, **mod**, =, /=, <, >, >=, <=, **sll**, **srl**, **rol**, **ror**, **sla**, **sra**

The following operations are defined for sfixed:

+, –, *, /, **rem**, **mod**, =, /=, <, >, >=, <=, **sll**, **srl**, **rol**, **ror**, **sla**, **sra**, **abs**, – (unary)

All of the binary operators are overloaded for REAL and INTEGER data types. In the case of a REAL, the range of the fixed-point number is used to convert the real number into fixed point before the operation is performed. In the case of an INTEGER, the number is converted into fixed point with the range of fixed'HIGH **downto** 0. Thus, the fixed-point operand must be of a format large enough to accommodate the converted input or a "vector-truncated" warning is produced. In these functions, overflow_style is set to fixed_saturate.

The overloaded definitions allow, as an example:

```
signal x: sfixed (4 downto -5);
signal y: real;
...
z := x + y;
```

In the case where an operation is performed that includes both a fixed-point number and an integer or real, the sizing rules are modified. For a real number, the real is converted to a fixed-point number that is the same size as the fixed-point argument. Thus, the preceding example is equivalent to:

```
z := x + sfixed(y, 4, -5);
```

result in a type of sfixed (5 **downto** –5) for z. A similar rule holds for integers.

Shift operators are functionally the same as those for NUMERIC_STD. An arithmetic shift (**sra** or **sla**) on an unsigned number is the same as a logical shift. An arithmetic shift on a signed number is the same as a logical shift if the number is shifted left, but replicates the sign bit if the number is shifted right.

The scalb function can be used to losslessly multiply or divide any number by a power of two, for example:

```
   constant half: ufixed (2 downto -2) := "00010"; -- 000.10
   variable two: ufixed (5 downto 0);
   variable someval: ufixed (5 downto -5);
begin
   two := scalb(half, 2); -- returns "00010.", or 2.0
   someval := resize (scalb (half, X), someval'high, someval'low);
```

All of the standard relational operators are implemented. The operators =, /=, <, >, >=, <= perform in a similar way to the NUMERIC_STD functions. If values of two different lengths are given, then the inputs are resized before the comparison is made.

The maximum and minimum functions do a comparison operation and return the appropriate value. These functions are overloaded for INTEGER and REAL parameters. The sizes of the parameters do not need to match. The output is resized to the maximum of the left index and minimum of the right index.

The find_leftmost and find_rightmost functions find the leftmost or rightmost occurrence of a given bit value in a fixed-point number and return the index of the occurrence. The functions are declared as:

```
function find_leftmost  (arg: ufixed; y: STD_ULOGIC) return INTEGER;
function find_rightmost (arg: ufixed; y: STD_ULOGIC) return INTEGER;
```

and similarly for sfixed parameters. The parameter y can be any STD_ULOGIC value. The functions use the "?=" operator to compare bits in arg with y, so strength of values is ignored. If the value is not found by the find_leftmost function, arg'low – 1 is returned. Similarly, if the value is not found by the find_rightmost function, arg'high + 1 is returned. Note that find_leftmost(arg, '1') for a ufixed parameter or for a positive sfixed parameter returns the integer log (base 2) of arg.

The To_01, To_X01, To_X01Z, To_UX01, and Is_X functions are similar to the STD_LOGIC_1164 and NUMERIC_STD functions of the same names.

Most synthesis tools do not support any I/O format other than std_logic_vector and std_logic. Thus, functions are included to convert between std_logic_vector and ufixed or sfixed, and vice versa, for example:

```
uf7_3 <= to_ufixed (slv7, uf7_3'high, uf7_3'low);
```

and

```
slv7 <= to_slv (uf7_3);
```

READ, WRITE, HREAD, HWRITE, OREAD, and OWRITE routines are also defined for fixed-point data types. A "." separator is added between the integer part and the fractional part of the fixed-point number. Therefore the result of to_ufixed (6.5, 4, –5) would be written as "00110.10000". This string can also be read back into a variable of type ufixed(4 **downto** –5).

The functions to_string, to_ostring, and to_hstring are also provided. These are very useful in assertion and report statements, for example:

```
assert x = y
   report to_string(x) & " /= " & to_string(y)
   severity error;
```

Alternatively, the numbers can be shown in real format:

```
assert x = y
   report to_string(to_real(x)) & " /= " & to_string(to_real(y))
   severity error;
```

In order to provide a measure of compatibility with tools commonly used to define DSP algorithms, the package provides the To_SFix and To_UFix conversion functions. These functions convert from a STD_LOGIC_VECTOR value to a ufixed or sfixed value, respectively. The index bounds for the result are described in terms of the vector length and the number of post-binary-point bits. For example, a DSP tool might describe an unsigned fixed-point number as ufix[14,10], which specifies a 14-bit word with a 10-bit fraction. This translates to the unsigned fixed-point type ufixed(3 **downto** –10). Similarly, sfix[14, 10] translates to the signed fixed-point type sfixed(3 **downto** –10).

G.4.6 Package generics

The fixed-point packages are defined by an uninstantiated package with generic constants, as follows:

```
library IEEE; ...
use IEEE.fixed_float_types.all;
package fixed_generic_pkg is
   generic (
      fixed_round_style    : fixed_round_style_type
         := fixed_round;
      fixed_overflow_style : fixed_overflow_style_type
         := fixed_saturate;
      fixed_guard_bits     : NATURAL := 3;
      no_warning           : BOOLEAN := FALSE
      );
   ...
```

Since it is an uninstantiated package, fixed_generic_pkg cannot be used directly. Rather, it must be instantiated and the instance used. The library IEEE contains a standard instance, named fixed_pkg, declared as:

```
library IEEE;
package fixed_pkg is new IEEE.fixed_generic_pkg
   generic map (
      fixed_round_style    => IEEE.fixed_float_types.fixed_round,
      fixed_overflow_style => IEEE.fixed_float_types.fixed_saturate,
      fixed_guard_bits     => 3,
      no_warning           => FALSE
      );
```

This is where the actual generics are specified. Note that the user can declare a separate instantiation of the fixed-point package if different defaults are required. For example, if an application does not require rounding (because it takes up too much logic), requires wrapping of numbers rather than saturation, requires no guard bits on divisions, and does not require “metavalue detected” warnings, the package may be instantiated as follows:

```
library IEEE;
package my_fixed_pkg is new IEEE.fixed_generic_pkg
   generic map (
      fixed_round_style    => IEEE.fixed_float_types.fixed_truncate,
      fixed_overflow_style => IEEE.fixed_float_types.fixed_wrap,
      fixed_guard_bits     => 0,
      no_warning           => TRUE
      );
```

This package instance can be analyzed and used in other design units. Note that the ufixed and sfixed types declared in the different package instances are distinct types, so type conversions may be needed to translate between them, as shown in the following example:

```
library IEEE; use IEEE.std_logic_1164.all, IEEE.fixed_pkg.all;
entity sin is
   port (arg: in ufixed (1 downto -16);
         clk, rst: in STD_ULOGIC;
         res: out ufixed (1 downto -11));
```

```
end entity sin;

architecture structure of sin is
   component fixed_sin is
      port (arg: in work.my_fixed_pkg.ufixed (1 downto -16);
            clk, rst: in STD_ULOGIC;
            res: out work.my_fixed_pkg.ufixed (1 downto -11));
   end component fixed_sin;
begin
   U1: component fixed_sin
      port map (arg => work.my_fixed_pkg.ufixed(arg), -- convert arg
                clk => clk, rst => rst,
                IEEE.fixed_pkg.ufixed (res) => res);
end architecture structure;
```

G.4.7 Issues

The fixed-point math packages are based on the NUMERIC_STD package and use signed and unsigned arithmetic from within that package. This makes them highly efficient because the NUMERIC_STD package is well supported by simulation and synthesis tools.

An ascending index range is treated as an error by the fixed-point routines. Thus, if a number is declared as ufixed(–1 **to** 5), an error will occur when the number is operated upon.

String literals also cause problems. For example, in the following:

```
z <= a + "011011";
```

the index range of the string literal is defined by VHDL rules to be INTEGER'left to INTEGER'left + 5. Infeasible index values such as these also cause errors to occur.

Care is required in cases such as the following:

```
   signal a: sfixed (3 downto -3);
   signal b: sfixed (2 downto -4);
begin
   b <= a;
```

In this example, the two vectors have the same length, and so the assignment is legal. However, the change in index range implies a shift in the position of the binary point, thus changing the value represented. For example, if a represents the value 6.5, after the assignment, b represents the value 3.25. Such direct assignments are only correct if the index ranges are the same. Otherwise, the resize function should be used.

G.4.8 Catalog of operations

G.4.8.1 Operators

"+" Adds two fixed-point numbers together, overloaded for REAL and INTEGER. See output sizing rules (G.4.3).

"–" Subtracts fixed-point numbers. Overloaded for REAL and INTEGER. See output sizing rules (G.4.3). Unary version (–var1) returns a value that is one bit larger than the input. Note that unary – is only implemented on sfixed.

"*" Multiply two fixed-point numbers together. Overloaded for REAL and INTEGER. See output sizing rules (G.4.3).

"/" Divides two fixed-point numbers. Overloaded for REAL and INTEGER. See output sizing rules (G.4.3). Uses 3 guard bits and rounds the result by default. If this is not the desired functionality, then use the divide function or modify the package generics.

"abs" Absolute value. Returns a result one bit larger than the input. The argument and result are both sfixed.

"mod" Modulo. Returns the signed remainder. See output sizing rules (G.4.3). Overloaded for REAL and INTEGER.

"rem" Remainder. Returns the unsigned remainder. See output sizing rules (G.4.3). Overloaded for REAL and INTEGER.

"sll" Shift left logical. Left argument is ufixed or sfixed, right argument is INTEGER. A negative right argument causes a logical right shift.

"srl" Shift right logical. Left argument is ufixed or sfixed, right argument is INTEGER. A negative right argument causes a logical left shift.

"rol" Rotate logical left. Left argument is ufixed or sfixed, right argument is INTEGER. A negative right argument causes a rotate right.

"ror" Rotate logical right. Left argument is ufixed or sfixed, right argument is INTEGER. A negative right argument causes a rotate left.

"sla" Shift left arithmetic. Left argument is ufixed or sfixed, right argument is INTEGER. A negative right argument causes right arithmetic shift. Note that a right arithmetic shift on an sfixed replicates the sign bit. A left shift does not replicate the least significant bit. Note also that “x **sla** int” will multiply (or divide) x by a power of 2.

"sra" Shift right arithmetic. Left argument is ufixed or sfixed, right argument is INTEGER. A negative right argument causes left arithmetic shift. Note that a right arithmetic shift on an sfixed replicates the sign bit. A left shift does not replicate the least significant bit. Note that “x **sra** int” will divide (or multiply) X by a power of 2.

"=" Equal. Overloaded for REAL and INTEGER. Returns FALSE if any 'X' is found. Integers are converted to fixed point with to_fixed (arg, max(a'high+1, 0), 0), Reals are converted with to_fixed (arg, a'high+1, a'low) and rounded.

"/=" Not equal. Overloaded for REAL and INTEGER. Returns TRUE if any 'X' is found. Integers are converted to fixed point with to_fixed (arg, max(a'high+1, 0), 0), reals are converted with to_fixed (arg, a'high+1, a'low) and rounded.

"<" Less than. Overloaded for REAL and INTEGER. Returns FALSE if any 'X' is found. Integers are converted to fixed point with to_fixed (arg, max(a'high+1, 0), 0), reals are converted with to_fixed (arg, a'high+1, a'low) and rounded.

">" Greater than. Overloaded for REAL and INTEGER. Returns FALSE if any 'X' is found. Integers are converted to fixed point with to_fixed (arg, max(a'high+1, 0), 0), reals are converted with to_fixed (arg, a'high+1, a'low) and rounded.

"<=" Less than or equal. Overloaded for REAL and INTEGER. Returns FALSE if any 'X' is found. Integers are converted to fixed point with to_fixed (arg, max(a'high+1, 0), 0), reals are converted with to_fixed (arg, a'high+1, a'low) and rounded.

">=" Greater than or equal. Overloaded for REAL and INTEGER. Returns FALSE if any 'X' is found. Integers are converted to fixed point with to_fixed (arg, max(a'high+1, 0), 0), reals are converted with to_fixed (arg, a'high+1, a'low) and rounded.

"?=" Performs an operation similar to the NUMERIC_STD "?=" function, but returns a STD_ULOGIC value.

"?/=" Performs an operation similar to the NUMERIC_STD "?=" function, but returns a STD_ULOGIC value.

"?<" Returns 'X' if a metavalue is in either number, '1' if L is less than R, otherwise '0'.

"?<=" Returns 'X' if a metavalue is in either number, '1' if L is less than or equal to R, otherwise '0'.

"?>" Returns 'X' if a metavalue is in either number, '1' if L is greater than R, otherwise '0'.

"?>=" Returns 'X' if a metavalue is in either number, '1' if L is greater than or equal to R, otherwise '0'.

"and" Logical and. Similar to the STD_LOGIC_1164 operators. Binary operators require operands to have the same index ranges. Index range of the result is the same as those of the operands.

"nand" Logical nand. Similar to the STD_LOGIC_1164 operators. Binary operators require operands to have the same index ranges. Index range of the result is the same as those of the operands.

"or" Logical or. Similar to the STD_LOGIC_1164 operators. Binary operators require operands to have the same index ranges. Index range of the result is the same as those of the operands.

"nor" Logical nor. Similar to the STD_LOGIC_1164 operators. Binary operators require operands to have the same index ranges. Index range of the result is the same as those of the operands.

"xor" Logical exclusive or. Similar to the STD_LOGIC_1164 operators. Binary operators require operands to have the same index ranges. Index range of the result is the same as those of the operands.

"xnor" Logical exclusive nor. Similar to the STD_LOGIC_1164 operators. Binary operators require operands to have the same index ranges. Index range of the result is the same as those of the operands.

"not" Logical not. Similar to the STD_LOGIC_1164 operator. Index range of the result is the same as that of the operand.

G.4.8.2 Functions

find_leftmost Find leftmost occurrence of a given bit value. Inputs: arg (ufixed or sfixed), y : std_ulogic. Returns the integer index of the first occurrence of y in the vector arg starting from the left. Arg'low–1 is returned if y is not found. Note that find_leftmost(arg, '1') for a ufixed parameter or for a positive sfixed parameter returns the integer log base 2 of the input arg.

find_rightmost Find rightmost occurrence of a given bit value. Inputs: arg (ufixed or sfixed), y : std_ulogic. Returns the integer index of the first occurrence of y in the vector arg starting from the right. Arg'high+1 is returned if y is not found.

divide Arithmetic divide. Functionally identical to the "/" operator, but with two extra parameters. Inputs: l, r (both ufixed or sfixed), parameters: guard_bits : NATURAL, round_style : fixed_round_style_type. See output sizing rules (G.4.3). Guard bits are extra bits that are added to the end of the divide routine to maintain precision when rounding. The round style is either fixed_round or fixed_truncate. If rounding is set to fixed_truncate, then the guard bits are ignored.

reciprocal Performs a 1/arg function. Inputs: arg (ufixed or sfixed), guard_bits : NATURAL, round_style : fixed_round_style_type. See output sizing rules (G.4.3). Guard bits are extra bits that are added to the end of the divide routine to maintain precision when rounding. The round style is either fixed_round or fixed_truncate. If rounding is set to fixed_truncate, then the guard bits are ignored.

remainder Arithmetic remainder. Inputs: l, r (both ufixed or sfixed), parameters: guard_bits : NATURAL, round_style : fixed_round_style_type. See output sizing rules (G.4.3). Guard bits are extra bits that are added to the end of the remainder routine to maintain precision when rounding. The round style is either fixed_round or fixed_truncate. If rounding is set to fixed_truncate, then the guard bits are ignored.

modulo Arithmetic modulo. Inputs: l, r (both ufixed or sfixed), Parameters: guard_bits : NATURAL, round_style : fixed_round_style_type. See output sizing rules (G.4.3). Guard bits are extra bits that are added to the end of the remainder routine to maintain precision when rounding. The round style is either fixed_round or fixed_truncate. If rounding is set to fixed_truncate, then the guard bits are ignored.

minimum Returns the minimum of the two inputs (both either ufixed or sfixed) by performing a ">" operation.

maximum Returns the maximum of the two inputs (both either ufixed or sfixed) by performing a ">" operation.

std_match Performs a NUMERIC_STD.STD_MATCH function (allows use of '–' values for the inputs).

add_carry	This procedure which takes in two parameters (L and R) as well as a carry in (C_IN). It has output parameters for a carry out (C_OUT) and a result of the same length as the combined width of L and R. Note that this routine can be used as an accumulator.
scalb	Inputs are ufixed or sfixed, with an INTEGER or signed input. The Scalb function moves the index of the fixed-point number, having the effect of multiplying or dividing by a power of two.

G.4.8.3 Conversion functions

Resize	Changes the size of a ufixed or sfixed (larger or smaller). Inputs: arg (ufixed or sfixed); left_index and right_index (INTEGER), or size_res (same type as arg). Other parameters: round_style, saturate_style. Output: resized ufixed or sfixed.	
To_ufixed	Converts to the ufixed type.	
	To_ufixed (std_ulogic_vector)	Inputs: arg (std_ulogic_vector); left_index and right_index (INTEGER), or size_res (ufixed). This function converts a std_ulogic_vector to a ufixed with the same width. A warning is produced if the width is incorrect.
	To_ufixed (unsigned)	Inputs: arg (unsigned); left_index and right_index (INTEGER), or size_res (ufixed). Other parameters: overflow_style, round_style. Converts an unsigned to a ufixed.
	To_ufixed (unsigned)	Inputs: arg (unsigned). Converts an unsigned to a ufixed of the same size with the left_index being arg'length–1 and the right_index being 0.
	To_ufixed (REAL)	Inputs: arg (REAL); left_index and right_index (INTEGER), or size_res (ufixed). Other parameters: overflow_style, round_style. Converts a REAL to a ufixed. If the input is negative, then an error occurs and 0 is returned.
	To_ufixed (INTEGER)	Inputs: arg (NATURAL); left_index and right_index (INTEGER), or size_res (ufixed). Other parameters: overflow_style, round_style. Converts an INTEGER to a ufixed.

To_sfixed	Converts to the sfixed type.	
	To_sfixed (std_ulogic_vector)	Inputs: arg (std_ulogic_vector); left_index and right_index (INTEGER), or size_res (sfixed). This function converts a std_ulogic_vector to an sfixed with the same width. A warning is produced if the width is incorrect.
	To_sfixed (signed)	Inputs: arg (signed); left_index and right_index (INTEGER), or size_res (sfixed). Other parameters: overflow_style, round_style. Converts a signed to an sfixed.
	To_sfixed (signed)	Inputs: arg (signed). Converts a signed to an sfixed of the same size with the left_index being arg'length–1 and the right_index being 0.
	To_sfixed (REAL)	Inputs: arg (REAL); left_index and right_index (INTEGER), or size_res (sfixed). Other parameters: overflow_style, round_style. Converts a REAL to an sfixed.
	To_sfixed (INTEGER)	Inputs: arg (INTEGER); left_index and right_index (INTEGER), or size_res (sfixed). Other parameters: overflow_style, round_style. Converts a INTEGER to an sfixed.
	To_sfixed (ufixed)	Inputs: arg (ufixed). Converts a ufixed into an sfixed by adding a sign bit.
To_unsigned	Inputs: arg (ufixed); and size (NATURAL), or size_res (unsigned). Other parameters: round_style, saturate_style. Converts a ufixed to an unsigned. This does not produce a “vector truncated” warning as the NUMERIC_STD functions do.	
To_signed	Inputs: arg (sfixed); and size (NATURAL), or size_res (signed). Other parameters: round_style, saturate_style. Converts an sfixed to a signed. This does not produce a “vector truncated” warning as the NUMERIC_STD functions do.	
To_real	Inputs: arg (ufixed or sfixed). Converts a fixed-point number to a real number.	
To_integer	Inputs: arg (ufixed or sfixed). Other parameters: round_style, saturate_style. Converts a fixed-point number to an integer.	
To_slv	Inputs: arg (ufixed or sfixed). Converts a fixed-point number to a std_logic_vector of the same length.	
To_std_logic_vector	Alias of to_slv.	
To_stdlogicvector	Alias of to_slv.	
To_sulv	Inputs: arg (ufixed or sfixed). Converts a fixed-point number to a std_ulogic_vector of the same length.	
To_std_ulogic_vector	Alias of to_sulv.	
To_stdulogicvector	Alias of to_sulv.	
To_01	Inputs s (ufixed or sfixed). Other parameters: XMAP: std_ulogic. Converts metavalues in the vector S to the XMAP state (defaults to 0).	
Is_X	Inputs arg (ufixed or sfixed). Returns a BOOLEAN which is TRUE if there are any metavalues in the vector arg.	
To_x01	Inputs arg (ufixed or sfixed). Converts any metavalues found in the vector arg to be 'X' , '0', or '1'.	

To_ux01	Inputs arg (ufixed or sfixed). Converts any metavalues found in the vector arg to be 'U', 'X' , '0', or 1'.
To_x01z	Inputs arg (ufixed or sfixed). Converts any metavalues found in the vector arg to be 'Z', 'X' , '0', or '1'.

G.4.8.4 Sizing functions

Each of these functions take as a parameter a character that describes the operation to be performed, as shown in Table G.3:

Table G.3—Operations described by characters

Character	Operation
'+'	"+"
'–'	"–"
'*'	"*"
'/'	"/", divide
'1'	reciprocal
'M', 'm'	"mod", modulo
'R', 'r'	"rem", remainder
'A', 'a'	"abs"
'N', 'n'	unary "–"
others	index

Ufixed_high	Inputs: left_index, right_index: INTEGER (bounds of the left argument) or size_res: ufixed; operation: character; left_index2, right_index2: INTEGER (bounds of the left argument) or size_res2: ufixed. This function is used to compute the high index bound of the result of an unsigned operation. Any values for the operation character other than those defined in Table G.3 cause the left_index to be returned.
Ufixed_low	Inputs: left_index, right_index: INTEGER (bounds of the left argument) or size_res: ufixed; operation: character; left_index2, right_index2: INTEGER (bounds of the left argument) or size_res2: ufixed. This function is used to compute the low index bound of the result of an unsigned operation. Any values for the operation character other than those defined in Table G.3 cause the left_index to be returned.
Sfixed_high	Inputs: left_index, right_index: INTEGER (bounds of the left argument) or size_res: ufixed; operation: character; left_index2, right_index2: INTEGER (bounds of the left argument) or size_res2: ufixed. This function is used to compute the high index bound of the result of a signed operation. Any values for the operation character other than those defined in Table G.3 cause the left_index to be returned.
Sfixed_low	Inputs: left_index, right_index: INTEGER (bounds of the left argument) or size_res: ufixed; operation: character; left_index2, right_index2: INTEGER (bounds of the left argument) or size_res2: ufixed. This function is used to compute the low index bound of the result of a signed operation. Any values for the operation character other than those defined in Table G.3 cause the left_index to be returned.
To_ufix	Similar to to_ufixed, but with NATURAL arguments representing the length of the result and the number of post-binary-point bits. Thus, for example, to_ufix ("00100", 5, 3) = 00.100, or 0.5.

To_sfix	Similar to to_sfixed, but with NATURAL arguments representing the length of the result and the number of post-binary-point bits. The sign bit is assumed to take an additional place beyond the specified length. Thus, for example, to_sfix("00100", 4, 3) = 00.100 or 0.5.
Ufix_high	Similar to ufixed_high, but with NATURAL arguments representing the length of the result and the number of post-binary-point bits.
Ufix_low	Similar to ufixed_low, but with NATURAL arguments representing the length of the result and the number of post-binary-point bits.
Sfix_high	Similar to sfixed_high, but with NATURAL arguments representing the length of the result and the number of post-binary-point bits.
Sfix_low	Similar to sfixed_low, but with NATURAL arguments representing the length of the result and the number of post-binary-point bits.

G.4.8.5 Textio functions

Write	Similar to the TEXTIO write procedure. Automatically inserts a binary point where needed. If the range of the input number does not include the 0 index, then the number is extended until it does before writing.
Read	Similar to the TEXTIO read procedure. If a binary point is encountered, then it is tested to ensure that it is in the correct place.
Bwrite	Alias for write.
Binary_write	Alias for write.
Bread	Alias for read.
Binary_read	Alias for read.
Owrite	Octal write. The pre- and post-binary-point parts of the number are written separately, with a binary point between them. Each side is padded to a multiple of 3 bits to form an octal digit.
Oread	Octal read. The number read is interpreted as separate pre- and post-binary-point parts, with an optional binary point between them. If a “.” is found in the input string, then the index is checked to ensure that the binary point is in the correct place.
Octal_write	Alias for owrite.
Octal_read	Alias for oread.
Hwrite	Hex write. The pre- and post-binary-point parts of the number are written separately, with a binary point between them. Each side is padded to a multiple of 4 bits to form a hex digit.
Hread	Hex read. The number read is interpreted as separate pre- and post-binary-point parts, with an optional binary point between them. If a “.” is found in the input string, then the index is checked to ensure that the binary point is in the correct place.
Hex_write	Alias for hwrite.
Hex_read	Alias for hread.
To_string	Returns a string that can be padded and left or right justified, for example: `assert a = 1.5 report "Result was " & to_string (a) severity error;`
To_bstring	Alias for to_string.
To_binary_string	Alias for to_string.

To_ostring	Similar to to_string, but returns an octal value with a binary point. The padding rules of the owrite procedure apply to this function.
To_octal_string	Alias for to_ostring.
To_hstring	Similar to to_string, but returns a hex value with a binary point. The padding rules of the hwrite procedure apply to this function.
To_hex_string	Alias for to_hstring.

From_string	Translates a string (with a binary point in it) to a fixed-point number. Some examples are:
	
	`signal a: ufixed (3 downto -3);`
	`begin`
	`a <= from_string ("0000.000", a'high, a'low);`
	`a <= from_string ("0001.000", a);`
	`a <= from_string ("0000.100"); -- Works only if`
	`-- size is exact.`
	
	Note that this is typically not synthesizable, as it uses the STRING type. A synthesizable alternative is “a <= "0000000";”.

From_bstring	Alias for from_string.
From_binary_string	Alias for from_string.
From_ostring	Same as from_string, but uses octal numbers. The oread padding rules apply in this function.
From_octal_string	Alias for from_ostring.
From_hstring	Same as from_string, but uses hex numbers. The hread padding rules apply in this function.
From_hex_string	Alias for from_hstring.

G.5 Using the floating-point package

While floating-point numbers are widely used in software applications, they are less common in custom hardware. This is because floating point takes up almost three times the hardware resources of fixed-point math. The advantage of floating point, however, is that relative precision is maintained across a wide dynamic range, whereas fixed-point numbers are limited to a smaller dynamic range with fixed absolute precision.

G.5.1 Floating-point numbers

Floating-point numbers are well defined by IEEE 754 (32 and 64 bit) and IEEE 854 (variable width) specifications. Floating point has been used in processors and intellectual property (IP) for years, and is a well-understood format. The format is a sign magnitude system, where the sign is processed separately from the magnitude.

There are many concepts in floating point that make it different from common signed and unsigned number notations. To illustrate, consider a 32-bit floating-point number:

```
S   EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF
31  30    23 22                    0
+/- exp.     fraction
```

Basically, a floating-point number comprises a sign bit (+ or –), a normalized exponent, and a fraction. To convert this number back into an integer, the following equation can be used:

S * 2**(exponent – exponent_base) * (1.0 + fraction/fraction_base)

where the "exponent_base" is 2**((maximum exponent/2) – 1), and "fraction_base" is the maximum possible fraction (unsigned) plus one. For example, using a 32-bit representation:

```
0 10000001 10100000000000000000000
```
= +1 * 2**(129 – 127) * (1.0 + 5242880/8388608) = +1 * 4.0 * 1.625 = 6.5

There are also "denormal numbers," which are numbers smaller than can be represented in this way. A denormal number is indicated with an exponent of 0. In this case, the term 1.0 is not added to the scaled fraction. For example:

```
1 00000000 10000000000000000000000
```
= –1 * 2**(– 126) * (4194304/8388608) = –1 * 2**(–126) * 0.5 = –2**(–127)

Next, there are several floating-point "constants":

```
0 00000000 00000000000000000000000
```
= +0.0

```
1 00000000 00000000000000000000000
```
= –0 (which = +0)

```
0 11111111 00000000000000000000000
```
= positive infinity

```
1 11111111 00000000000000000000000
```
= negative infinity

A number with an infinite (all ones) exponent and anything other than an all-zero fraction is said to be a NaN, or "Not a Number." There are two types of NaN: signaling and non-signaling. The floating-point package defines a NaN with a fraction whose most significant bit is 1 to be a signaling NAN and any other NaN to be a quiet NaN.

In summary, a floating-point number falls into one of the following classes (or states):

- nan: Signaling NaN
- quiet_nan: Quiet NaN
- neg_inf: Negative infinity
- neg_normal: Negative normalized non-zero
- neg_denormal: Negative denormalized
- neg_zero: –0.0
- pos_zero: +0.0
- pos_denormal: Positive denormalized
- pos_normal: Positive normalized non-zero
- pos_inf: Positive infinity
- isx: at least one input is unknown

These states correspond to enumeration values of the type valid_fpstate defined in the package, and are used to examine and create numbers needed for floating-point operations. The state isx is included to indicate the presence of one or more metavalues ('X', 'U', and so on) in a floating-point number. Any arithmetic operation on such a number will return a number with all bits 'X'.

The package also defines functions that return constant values represented in specified sizes:

- zerofp: +0.0

— nanfp: Signaling NaN
— qnanfp: Quiet NaN
— pos_inffp: Positive infinity
— neginf_fp: Negative infinity
— neg_zerofp: –0.0

Rounding can take four different forms:

— round_nearest: Round to nearest
— round_inf: Round toward positive infinity
— round_neginf: Round toward negative infinity
— round_zero: Round toward zero (truncate)

These forms correspond to enumeration values of the type round_type defined in the package IEEE.fixed_float_types. Parameters of the type control rounding behavior. In the case of rounding to the nearest value, if the remainder is exactly ½, the result is rounded so that the least significant bit is 0. The implementation of this form of rounding requires two comparison operations, but they can be consolidated. Rounding toward negative infinity rounds down, and rounding toward positive infinity rounds up. Rounding toward zero simply truncates the remainder, with no actual rounding.

G.5.2 Use model

An example of use of the floating-point package is:

```
   use IEEE.float_pkg.all;
   ...
   signal x, y, z: float (5 downto -10);
begin
   y <= to_float (3.1415, y); -- Uses y for sizing only
   z <= "0011101010101010";   -- 1/3
   x <= z + y;
```

The package defines three floating-point types:

— float32: 32-bit IEEE 754 single precision floating point
— float64: 64-bit IEEE 754 double precision floating point
— float128: 128-bit IEEE 854 extended precision floating point

The package also allows specification of a custom floating-point width by constraining the float type, as shown in the preceding example.

The 32-bit floating-point type is defined as follows:

subtype float32 **is** float (8 **downto** –23);

A negative index is used to separate the fraction part of the floating-point number from the exponent. The top bit ('high) is the sign bit, the next bits ('high–1 **downto** 0) are the exponent, and the bits with negative indices (–1 **downto** 'low) are the fraction. Thus, for a 32-bit representation, the number is represented as follows:

```
0 00000000 00000000000000000000000
8 7      0 -1                  -23
± exp.      fraction
```

where the sign is bit 8, the exponent is contained in bits 7 down to 0 (8 bits, with bit 7 being the most significant), and the mantissa is contained in bits –1 down to –23 (32 – 8 – 1 = 23 bits, where bit –1 is the most significant).

The negative index format turns out to be a very natural format for the floating-point number, as the fraction is always assumed to be a number between 1.0 and 2.0 (unless the number is denormalized). Thus, the implied "1.0" can be assumed on the positive side of the index, and the negative side represents a fraction of less than one. The format is similar to that used in the fixed-point package, where everything to the right of the zero index is assumed to be less than 1.0.

Valid values for float_exponent_width and float_fraction_width are 3 or more. Thus, the smallest (width-wise) number that can be represented is float (3 **downto** –3), a 7-bit floating-point number.

The base type defined in the package is unresolved_float (aliased to u_float). The type float is a subtype of unresolved_float, with resolved elements. The operations defined in the package can be used with either type interchangeably. The subtypes float32, float64, and float128 are subtypes of float with specified index ranges. The package also defines subtypes unresolved_float32 (aliased to u_float32), unresolved_float64 (aliased to u_float64), and unresolved_float128 (aliased to u_float128) as subtypes of unresolved_float with specified index ranges.

Operators for all of the standard math and compare operations are defined in this package. In the float_pkg package, these operators implement all aspects of IEEE floating-point operations. For most designs, full IEEE support is not necessary. Thus, functions have been created that allow a design to be parameterized, for example:

```
x <= add (l => z, r => y,
          denormalize => FALSE, -- turn off denormal numbers
                                -- (default=TRUE)
          check_error => FALSE, -- turn off NaN and overflow checks
                                -- (default=TRUE)
          round_style => round_zero, -- truncate
                                   -- (default=round_nearest)
          guard_bits => 0);     -- extra bits to maintain precision
                                -- (default=3)
```

The add function performs just like the + operator; however, it allows the user the flexibility needed for hardware synthesis. Other similar functions are subtract (–), multiply (*), divide (/), modulo (**mod**), and remainder (**rem**). All of these operators and functions assume that both of the inputs are the same width. Other functions with similar parameters are reciprocal (1/x) and dividebyp2 (divide by a power of 2). The **abs** and unary – operators need no parameters, as they only affect the sign of the floating-point number.

Comparison operators work similarly; however there is only one extra parameter for these functions, namely, the check_error parameter, which allows NaN and infinity testing to be turned off for the comparison. These functions are called EQ (=), NE (/=), LT (<), GT (>), GE (>=), and LE (<=).

Conversion functions also work in a similar manner. Functions named to_float are available to convert the types REAL, INTEGER, signed, unsigned, ufixed, and sfixed. All of these functions take as parameters either the exponent_width and fraction_width, or a size_res input, which uses the input value for its size only. The functions to_real, to_integer, to_signed, to_unsigned, to_ufixed, and to_sfixed are also overloaded in the package with both size and size_res inputs. Further, there is a similar resize function to convert from

one float size to another. Note that, as in the fixed_pkg package, an ascending index range (specified with **to**) for a float type is illegal.

The package includes a number of functions recommended by IEEE Std 754-1985 and IEEE Std 854-1987. They are described in G.5.4.4.

Two functions, named break_number and normalize, are also provided. Break_number takes a floating-point number and returns a SIGNED exponent (biased by –1), a ufixed fixed-point fraction, and a std_ulogic sign. Normalize takes a SIGNED exponent, a fixed-point fraction, and a sign and returns a floating-point number. These functions are useful for operating on the fraction of a floating-point number without having to perform the shifts on every operation.

To_slv (aliased as to_std_logic_vector and to_StdLogicVector) and to_float are used to convert between std_logic_vector and floating-point types. These may be used on the interfaces of designs. The result of to_slv is a std_logic_vector with the length of the input floating-point type.

Procedures for reading and writing floating-point numbers are also included in the package. Procedures read, write, oread, owrite (octal), bread, bwrite (binary), hread, and hwrite (hex) are defined. To_string, to_ostring, and to_hstring are also provided for string results. Floating-point numbers are written in a format such as “0:000:000” (for a 7-bit number). They can be read as a simple string of bits, or with a “.” or “:” separator.

The following example illustrates use of the package:

```
library IEEE; use IEEE.std_logic_1164.all;
entity xxx is
   port (a, b: in std_logic_vector (31 downto 0);
         sum: out std_logic_vector (31 downto 0);
         clk, reset: in std_ulogic);
end entity xxx;

use IEEE.float_pkg.all;
architecture RTL of xxx is
   signal afp, bfp, sumfp: float32;
begin
   afp <= to_float (a, afp'high, -afp'low); -- SLV to float, with bounds
   bfp <= to_float (b, bfp); -- SLV to float, using bfp'range
   addreg : process (clk, reset) is
   begin
      if reset = '1' then
         sumfp <= (others => '0');
      elsif rising_edge (clk) then
         sumfp <= afp + bfp;
         -- this is the same as saying:
         --   sumfp <= add (l => afp, r => bfp,
         --                 round_style => round_nearest,
         --                                   -- best, but most hardware
         --                 guard_bits => 3, -- Use 3 guard bits,
         --                                   -- best for round_nearest
         --                 check_error => TRUE,
         --                                   -- NaN processing turned on
         --                 denormalize => TRUE);
         --                                   -- Turn on denormal numbers
      end if;
```

```
   end process addreg;
   sum <= to_slv (sumfp);
end architecture xxx;
```

G.5.3 Package generics

Several aspects of floating-point arithmetic can take up a great deal of hardware. Depending on the application, not all aspects are needed, so the float_generic_pkg package is designed using generic constants to allow choice among aspects. The float_generic_pkg cannot be used directly, but must first be instantiated to provide actual values for the generic constants. The declaration of the float_generic_pkg is:

```
library IEEE; ...
use IEEE.fixed_float_types.all; ...
package float_generic_pkg is
   generic (
      float_exponent_width : NATURAL    := 8;
      float_fraction_width : NATURAL    := 23;
      float_round_style    : round_type := round_nearest;
      float_denormalize    : BOOLEAN    := TRUE;
      float_check_error    : BOOLEAN    := TRUE;
      float_guard_bits     : NATURAL    := 3;
      no_warning           : BOOLEAN    := FALSE;
      package fixed_pkg is new IEEE.fixed_generic_pkg
                              generic map (<>)
      );
   ...
```

The generic constants are used as follows:

- float_exponent_width: Default for conversion routines. For example, the value for a 32-bit floating-point number would be 8.
- float_fraction_width: Default for conversion routines. For example, the value for a 32-bit floating-point number would be 23.
- float_round_style: Specifies the rounding style to be used, as described in G.5.1.
- float_denormalize: Activates (TRUE) or deactivates (FALSE) use of denormal numbers.
- float_check_error: Activates (TRUE) or deactivates (FALSE) NaN and infinity processing. With processing activated, checks are done at the beginning of every operation. If checks have been done previously, processing does not need to be repeated for each operation.
- float_guard_bits: Specifies is the number of extra bits used in each operation to maintain precision. If the number of guard bits is zero, then rounding is automatically turned off.
- no_warning: Deactivates (TRUE) or activates (FALSE) “metavalue” warnings.
- fixed_pkg: The package defining fixed-point types for conversion functions.

There is also a standard instantiation, float_pkg, with actual values for the generics, defined as:

```
library IEEE;
package float_pkg is new IEEE.float_generic_pkg
   generic map (
      float_exponent_width => 8,
      float_fraction_width => 23,
      float_round_style => IEEE.fixed_float_types.round_nearest,
      float_denormalize => TRUE,
```



```
      float_check_error => TRUE,
      float_guard_bits => 3,
      no_warning => FALSE,
      fixed_pkg => IEEE.fixed_pkg
      );
```

Note that the user can can declare a separate instantiation of the floating-point package if different actual generics are required. For example, if an application does not require rounding (because it takes up too much logic), requires 17-bit floating-point numbers with only 5 bits of exponent, does not require denormal numbers or NaN and infinity processing, and does not require "metavalue detected" warnings, the package may be instantiated as follows:

```
library IEEE;
package my_float_pkg is new IEEE.float_generic_pkg
   generic map (
      float_exponent_width => 5,    -- 5 bits of exponent
      float_fraction_width => 11,   -- Default will be
                                    -- float(5 downto -11)
      float_round_style => IEEE.fixed_float_types.round_zero,
                                    -- Truncate, don't round
      float_denormalize => FALSE,   -- no denormal numbers
      float_guard_bits => 0,        -- Unused by round_zero, set to 0
      float_check_error => FALSE,   -- Turn NaN and overflow off
      no_warning => TRUE,           -- turn warnings off
      fixed_pkg => WORK.my_fixed_pkg
      );
```

This package instance can be analyzed and used in other design units. Those design units can include a use clause such as "**use** work.my_float_pkg.**all**;" to make the floating-point function visible. Note that the types declared in the different package instances are distinct types, so type conversions may be needed to translate between them, as shown in the following example:

```
use IEEE.float_pkg.all, IEEE.std_logic_1164.all;
entity sin is
   port (arg: in float (5 downto -11);
         clk, rst: in std_ulogic;
         res: out float (5 downto -11));
end entity sin;

architecture structure of sin is
   component float_sin is
      port (arg: in work.my_float_pkg.float (5 downto -11);
            clk, rst: in std_ulogic;
            res: out work.my_float_pkg.float (5 downto -11));
   end component fixed_sin;
begin
   U1: component float_sin
      port map (arg => work.my_float_pkg.float(arg), -- convert arg
                clk => clk, rst => rst,
                IEEE.float_pkg.float (res) => res);
end architecture structure;
```

G.5.4 Catalog of operations

G.5.4.1 Operators

"+" Add two floating-point numbers together. Overloaded for REAL and INTEGER. In float_pkg, rounding is set to round_nearest, 3 guard bits are used, and denormal number and NaN processing are turned on. If this is not the desired functionality, use the add function. Will accept floating-point numbers of any valid width on either input.

"–" Subtracts floating-point numbers. Overloaded for REAL and INTEGER. In float_pkg, rounding is set to round_nearest, 3 guard bits are used, and denormal number and NaN processing are turned on. If this is not the desired functionality, use the subtract function. Will accept floating-point numbers of any valid width on either input.

"*" Multiply two floating-point numbers together. Overloaded for REAL and INTEGER. In float_pkg, rounding is set to round_nearest, 3 guard bits are used, and denormal number and NaN processing are turned on. If this is not the desired functionality, use the multiply function. Will accept floating-point numbers of any valid width on either input.

"/" Divides two floating-point numbers. Overloaded for REAL and INTEGER. In float_pkg, rounding is set to round_nearest, 3 guard bits are used, and denormal number and NaN processing are turned on. If this is not the desired functionality, use the divide function. Will accept floating-point numbers of any valid width on either input.

"abs" Absolute value. Changes only the sign bit.

"–" Unary minus. Changes only the sign bit.

"mod" Modulo. Overloaded for REAL and INTEGER. In float_pkg, rounding is set to round_nearest, 3 guard bits are used, and denormal number and NaN processing are turned on. If this is not the desired functionality, use the modulo function. Will accept floating-point numbers of any valid width on either input.

"rem" Remainder. Overloaded for REAL and INTEGER. In float_pkg, rounding is set to round_nearest, 3 guard bits are used, and denormal number and NaN processing are turned on. If this is not the desired functionality, use the remainder function. Will accept floating-point numbers of any valid width on either input.

"=" Equal. Overloaded for REAL and INTEGER. In float_pkg, NaN processing is turned on. If this is not the desired functionality, then use the eq function.

"/=" Not equal. Overloaded for REAL and INTEGER. In float_pkg, NaN processing is turned on. If this is not the desired functionality, then use the ne function.

"<" Less than. Overloaded for REAL and INTEGER. In float_pkg, NaN processing is turned on. If this is not the desired functionality, then use the lt function.

">" Greater than. Overloaded for REAL and INTEGER. In float_pkg, NaN processing is turned on. If this is not the desired functionality, then use the gt function.

"<=" Less than or equal to. Overloaded for REAL and INTEGER. In float_pkg, NaN processing is turned on. If this is not the desired functionality, then use the le function.

">=" Greater than or equal to. Overloaded for REAL and INTEGER. In float_pkg, NaN processing is turned on. If this is not the desired functionality, then use the ge function.

"?=" Similar to "=", but returns a STD_ULOGIC value.

"?/=" Similar to "/=", but returns a STD_ULOGIC value.

"?<" Similar to "<", but returns a STD_ULOGIC value.

"?>" Similar to ">", but returns a STD_ULOGIC value.

"?<=" Similar to "<=", but returns a STD_ULOGIC value.

"?>=" Similar to ">=", but returns a STD_ULOGIC value.

"and" Logical and. Similar to the STD_LOGIC_1164 operators.

"nand" Logical nand. Similar to the STD_LOGIC_1164 operators.

"or" Logical or. Similar to the STD_LOGIC_1164 operators.

"nor" Logical nor. Similar to the STD_LOGIC_1164 operators.

"xor" Logical exclusive or. Similar to the STD_LOGIC_1164 operators.

"xnor" Logical exclusive nor. Similar to the STD_LOGIC_1164 operators.

"not" Logical not. Similar to the STD_LOGIC_1164 operator.

G.5.4.2 Functions

add	The add function is similar to the "+" operator; however, it allows the user to vary all of the parameters.
subtract	The subtract function is similar to the "–" operator; however, it allows the user to vary all of the parameters.
multiply	The multiply function is similar to the "*" operator; however, it allows the user to vary all of the parameters.
divide	The divide function is similar to the "/" operator; however, it allows the user to vary all of the parameters.
remainder	The remainder function is similar to the "rem" operator; however, it allows the user to vary all of the parameters.
modulo	The modulo function is similar to the "mod" operator; however, it allows the user to vary all of the parameters.
reciprocal	Returns 1/arg. Inputs: l, r: float; round_style: round_type; guard: NATURAL; check_error: BOOLEAN; denormalize: BOOLEAN. Works similarly to the divide function.
dividebyp2	Divide by a power of two. Inputs: l, r: float; round_style: round_type; guard: NATURAL; check_error: BOOLEAN; denormalize: BOOLEAN. Takes the exponent from R and multiplies L by that amount. Returns an error if R is not a power of 2.
mac	Multiply accumulate. Inputs: l, r, c: float; round_style: round_type; guard: NATURAL; check_error: BOOLEAN; denormalize: BOOLEAN. Performs the function L*R+C. The addition stage is integrated into the multiplier stage; thus, this operation takes less logic than separate calls to multiply and add.
sqrt	Square root. Inputs: arg: float; round_style: round_type; guard: NATURAL; check_error: BOOLEAN; denormalize: BOOLEAN. Returns the square root of arg, as defined by IEEE Std 754-1985.
Is_Negative	Returns TRUE if if the floating-point number is negative, or FALSE otherwise.
eq	The eq function is similar to the "=" operator; however, it allows the user to turn NaN processing is on or off.
ne	The ne function is similar to the "/=" operator; however, it allows the user to turn NaN processing is on or off.
lt	The lt function is similar to the "<" operator; however, it allows the user to turn NaN processing is on or off.
gt	The gt function is similar to the ">" operator; however, it allows the user to turn NaN processing is on or off.

le	The le function is similar to the "<=" operator; however, it allows the user to turn NaN processing is on or off.
ge	The ge function is similar to the ">=" operator; however, it allows the user to turn NaN processing is on or off.
std_match	Same as the NUMERIC_STD std_match function. Overloaded for type float.
maximum	Returns the larger of two numbers.
minimum	Returns the smaller of two numbers.

G.5.4.3 Conversion functions

Resize	Changes the size of a float (larger or smaller). Inputs: arg (float); exponent_width and fraction_width (NATURAL), or size_res; round_style: round_type; Check_error: BOOLEAN; denormalize_in: BOOLEAN; denormalize: BOOLEAN. In this function, denormalize_in is TRUE if the input number can be denormal, and denormalize is TRUE if the output number can be denormal.	
To_slv	Inputs: arg (float). Converts a floating-point number to a std_logic_vector of the same length.	
To_std_logic_vector	Alias for to_slv.	
To_stdlogicvector	Alias for to_slv.	
To_sulv	Inputs: arg (float). Converts a floating-point number to a std_ulogic_vector of the same length.	
To_std_ulogic_vector	Alias for to_sulv.	
To_stdulogicvector	Alias for to_sulv.	
To_float	Converts to the float type. The default size returned by these functions is set by float_exponent_width and float_fraction_width.	
	To_float (std_logic_vector)	Std_logic_vector to float. Inputs: arg (std_logic_vector); exponent_width and fraction_width (NATURAL), or size_res (float).
	To_float (INTEGER)	Integer to float. Inputs: arg (INTEGER); exponent_width and fraction_width (NATURAL), or size_res (float); round_style: round_type.
	To_float (REAL)	Real to float. Inputs: arg (REAL); exponent_width and fraction_width (NATURAL), or size_res (float); round_style: round_type; denormalize: BOOLEAN.
	To_float(ufixed)	Ufixed to float. Inputs: arg(ufixed); exponent_width and fraction_width (NATURAL), or size_res (float); round_style: round_type; denormalize: BOOLEAN.
	To_float(sfixed)	Sfixed to float. Inputs: arg(sfixed); exponent_width and fraction_width (NATURAL), or size_res (float); round_style: round_type; denormalize: BOOLEAN.
	To_float (signed)	Signed to float. Inputs: arg (signed); exponent_width and fraction_width (NATURAL), or size_res (float); round_style: round_type.
	To_float (unsigned)	Unsigned to float. Inputs: arg (signed); exponent_width and fraction_width (NATURAL), or size_res (float); round_style: round_type.

To_unsigned	Float to unsigned. Inputs: arg (float); size: NATURAL. Parameters: round_style: round_type; check_error: BOOLEAN. This does not produce a “vector truncated” warning as the NUMERIC_STD functions do. Returns a zero if the number is negative. Returns a saturated value if the input is too big.
To_signed	Float to signed. Inputs: arg (float); size: NATURAL. Parameters: round_style: round_type; check_error: BOOLEAN. This does not produce a “vector truncated” warning as the NUMERIC_STD functions do. Returns a saturated value if the number is too big.
To_ufixed	Float to ufixed. Inputs: arg (float); left_index and right_index (NATURAL), or size_res (ufixed). Parameters overflow_style: BOOLEAN; round_style: BOOLEAN; check_error: BOOLEAN; and denormalize: BOOLEAN.
To_sfixed	Float to sfixed. Inputs: arg (float); left_index and right_index (NATURAL), or size_res (ufixed). Parameters overflow_style: BOOLEAN; round_style: BOOLEAN; check_error: BOOLEAN; and denormalize: BOOLEAN.
To_real	Float to REAL. inputs: arg (float). Parameters: check_error: BOOLEAN; denormalize: BOOLEAN.
To_integer	Float to integer. inputs: arg (float). Parameters: round_style: round_type; check_error: BOOLEAN.
realtobits	Inputs: arg (REAL). Converts a real number to a std_ulogic_vector in the same format as a float64 floating-point number.
bitstoreal	Inputs: arg (std_ulogic_vector). Converts a std_ulogic_vector in the same format as a float64 floating-point number to a real number.
To_01	Inputs (arg: float). Parameters: xmap: std_ulogic. Converts metavalues in the vector arg to the xmap state (defaults to '0').
Is_X	Inputs (arg: float). Returns a BOOLEAN which is TRUE if there are any metavalues in the vector arg.
To_x01	Inputs (arg: float). Converts any metavalues found in the vector arg to be 'X' , and non-metavalues to '0' or '1'.
To_x01z	Inputs (arg: float). Converts any metavalues other than 'Z' found in the vector arg to be 'X' , and non-metavalues to '0' or '1'.
To_ux01	Inputs (arg: float). Converts any metavalues other than 'U' found in the vector arg to be 'X' , and non-metavalues to '0' or '1'.
Break_number	Procedure to break a floating-point number into its parts. Inputs: arg: float; denormalize: BOOLEAN; check_error: BOOLEAN. Output: fract: unsigned or ufixed fraction (with a '1' in the most significant bit); expon: the signed exponent (biased by –1, so add 1 to get the true exponent); sign: the sign bit.
Normalize	Function to take a fixed-point number and an exponent and return a floating-point number. Inputs: fract: ufixed; expon: signed (assumed to be biased by –1); sign: std_ulogic. Parameters: exponent_width and fraction_width (NATURAL), or size_res (float); round_style: round_type; denormalize: BOOLEAN; nguard: NATURAL. There is also a version of this function in which fract is an unsigned.

G.5.4.4 IEEE 754 and IEEE 854 recommended functions and predicates

copysign(x, y)	Returns x with the sign of y.
scalb(y, n)	Returns y*(2**n) (where n is an INTEGER or SIGNED) without computing 2**n.
logb(x)	Returns the unbiased exponent of x.

nextafter(x, y)	Returns the next representable number after x in the direction of y.
finite(x)	BOOLEAN, TRUE if X is not positive or negative infinity
isnan(x)	BOOLEAN, TRUE if X is a signaling or quiet NaN.
unordered(x, y)	BOOLEAN, returns TRUE of either x or y are some type of NaN.
classfp	Find the classification of a floating-point number. Inputs: arg (float). Returns a value of the type valid_fpstate. Note that IEEE Std 754-1985 and IEEE Std 854-1987 recommend the name “class” for this function. However, the floating-point package calls the function “classfp” to avoid conflict with “class” as a reserved word in a future extension of VHDL.

G.5.4.5 Functions returning constants

For each of the following, parameters are exponent_width and fraction_width, or size_res. The default size is set by the float_exponent_width and float_fraction_width generics.

zerofp	Returns a floating-point positive zero.
nanfp	Returns a floating-point signaling NaN.
qnanfp	Returns a floating-point quiet NaN.
pos_inffp	Returns a floating-point positive infinity.
neg_inffp	Returns a floating-point negative infinity.
neg_zerofp	Returns a floating-point negative zero (which by definition is equal to a floating-point positive zero).

G.5.4.6 Textio functions

write	Similar to the TEXTIO write procedure. Automatically puts in a “:” after the sign and the exponent.
read	Similar to the TEXTIO read procedure. If a decimal point or colon is encountered, then it is tested to be sure that it is in the correct place.
bwrite	Alias for write.
binary_write	Alias for write.
bread	Alias for read.
binary_read	Alias for read.
owrite	Octal write. If the number of bits is not divisible by three, then padding bits are added.
octal_write	Alias for owrite.
oread	Octal read. If the number of bits to be read is not divisible by three, then the number read is resized to fit.
octal_read	Alias for oread.
hwrite	Hex write. If the number of bits is not divisible by four, then padding bits are added.
hex_write	Alias for hwrite.

hread	Hex read. If the number of bits to be read is not divisible by four, then the number read is resized to fit.
hex_read	Alias for hread.

to_string	Returns a string that can be padded and left or right justified, for example:
	
	`assert (a = 1.5) report "Result was " & to_string (a)`
	`  severity error;`

to_bstring	Alias for to_string.
to_binary_string	Alias for to_string.
to_ostring	Similar to to_string, but returns a padded octal value.
to_octal_string	Alias for to_ostring.
to_hstring	Similar to to_string, but returns a padded hex value.
to_hex_string	Alias for to_hstring.

from_string	Allows translation of a string (with a binary point in it) into a floating-point number, for example:
	
	`  signal a: float (3 downto -3);`
	`begin`
	`  a <= from_string ("0000.000", a'high, -a'low);`
	`  a <= from_string ("0001.000", a);`
	
	Note that this is typically not synthesizable (as it uses the type string). An alternative assignment that is synthesizable is “A <= "0000000";”.

from_bstring	Alias for from_string.
from_binary_string	Alias for from_string.
from_ostring	Same as from_string, but uses octal numbers.
from_octal_string	Alias for from_ostring.
from_hstring	Same as from_string, but uses hex numbers.
from_hex_string	Alias for from_hstring.

Annex H

(informative)

Guide to use of protect directives

H.1 General

The protect tool directives described in Clause 24 allow authors of VHDL descriptions (so called *intellectual property,* or IP) to provide IP to users in such a way that the users cannot read the source text of the IP. The protect tool directives provide some underlying mechanisms for such protected IP exchange. This annex provides a guide to how those mechanisms may be used to ensure that IP exchange is secure and not subject to compromise. Note, however, that once IP has been delivered to a user's tool, the strength of protection against disclosure of the IP is entirely dependent on the tool.

The protect tool directives are used to form a cryptographic protocol in which IP is sent from the author to one or more user's tools, with the users considered untrusted third parties. Cryptographic protocols can be constructed to support the following use cases, among others:

— Delivery of IP from an author to any instance of a given decryption tool, and not for use on other decryption tools
— Delivery of IP from an author to a specific instance of a given decryption tool, and not for use on other instances of that decryption tool or any other decryption tool
— Delivery of IP from an author to a specific user for decryption by any of that user's decryption tools, and not for use by other users
— Delivery of IP from an author to several specific instances of a given decryption tool, and not for use on other instances of that decryption tool or any other decryption tool
— Delivery of IP from an author to several specific users for decryption by any of those users' decryption tools, and not for use by other users
— Use by a decryption tool of IP delivered by several authors
— Use by a user of IP delivered by several authors

Central to implementation of these use cases is embedding of appropriate encryption keys in tools. For example, decryption of IP can be limited to a specific instance of a given tool by embedding a given key in that instance only. Decryption can be limited to any instance of a given tool by embedding a given key in each instance, and not in any other tools. Decryption can be limited to a given user by providing that user with a key to be embedded in the user's tools.

The way in which keys may be embedded in tools and exchanged among authors, users, and tools is not specified by this standard. Nonetheless, secure exchange of keys is an integral part of any cryptographic protocol. This is discussed further in H.5. First, however, follows a discussion of various use cases, assuming the necessary keys are in place.

H.2 Simple protection envelopes

H.2.1 Symmetric cipher and secret key

The simplest form of IP delivery involves a symmetric cipher using a secret key shared by the IP author and the decryption tool. The author forms an encryption envelope in which is specified the symmetric cipher and

the secret key to use. For example, the following encryption envelope specifies the AES symmetric cipher using a secret key owned by a given user. Both the encrypting tool and the decrypting tool are assumed to have access to the secret key.

```
`protect data_keyowner="ACME IP User", data_method="aes192-cbc"
`protect begin
IP source text
...
`protect end
```

The encryption tool generates a decryption envelope specifying the cipher and secret key:

```
`protect begin_protected
`protect encrypt_agent="Encryptomatic", encrypt_agent_info="2.3.4a"
`protect data_keyowner="ACME IP User", data_method="aes192-cbc"
`protect encoding = (enctype="base64", line_length=40, bytes=4006), data_block
encoded encrypted IP
...
`protect end_protected
```

The user's decryption tool uses the key owner information to access the secret key and decrypts the IP using the AES cipher with that key.

H.2.2 Default cipher and key

The rules for protection envelopes allow specification of the cipher and key to be omitted, in which case, the cipher and key are chosen in an implementation-defined manner. One possible way for this mechanism to be used is to imply encryption using a default cipher with a key provided by the tool vendor and embedded in the encryption and decryption tools. For example, an encryption envelope using this scheme contains only the directives bracketing the IP source code:

```
`protect begin
IP source text
...
`protect end
```

The encryption tool includes information about the cipher and key it chooses in the decryption envelope:

```
`protect begin_protected
`protect encrypt_agent="Encryptomatic", encrypt_agent_info="2.3.4a"
`protect data_keyowner="Electrowizz Co", data_keyname="crypto-101"
`protect data_method="des-cbc"
`protect encoding = (enctype="base64", line_length=40, bytes=4006), data_block
encoded encrypted IP
...
`protect end_protected
```

H.2.3 Specification of encoding method

An encryption envelope may also include specification of the encoding method to use for encrypted information in the decryption envelope produced by the encryption tool. In the absence of an encoding directive in the encryption envelope, the encryption tool chooses a method, as in the preceding example. An example including an encoding directive is:

```
`protect data_keyowner="ACME IP User", data_method="aes192-cbc"
```



```
`protect encoding = (enctype="quoted-printable", line_length=60)
`protect begin
IP source text
...
`protect end
```

H.3 Digital envelopes

H.3.1 Encryption for a single user

A digital envelope allows an author to provide IP to one or more selected tools or users. A common use case is encryption using an asymmetric cipher for a single user's decryption tool. The private key is embedded in the user's tool, and the public key is published. While the IP could be encrypted using the public key, using a simple decryption envelope as described in H.2, asymmetric encryption is computationally expensive. Instead, the author can specify that a digital envelope be used, with a symmetric cipher used to encrypt the IP, and the key for the symmetric cipher encrypted using the decryption tool's public key. The encryption envelope is specified as follows:

```
`protect key_keyowner="ACME IP User", key_method="rsa", key_block
`protect data_method="aes192-cbc"
`protect begin
IP source text
...
`protect end
```

In this case, the presence of the key keyowner and key method directives specifies that the encryption tool use a digital envelope. The data method directive specifies the particular symmetric for encrypting the IP. The encryption tool chooses a session key (that is, the key used to encrypt and decrypt the IP). In the decryption envelope, it includes a key block containing the encrypted session key and a data block containing the encrypted IP, as follows:

```
`protect begin_protected
`protect encrypt_agent="Encryptomatic", encrypt_agent_info="2.3.4a"
`protect key_keyowner="ACME IP User", key_method="rsa"
`protect encoding = (enctype="base64", line_length=40, bytes=256), key_block
encoded encrypted session key
...
`protect data_method="aes192-cbc"
`protect encoding = (enctype="base64", line_length=40, bytes=4006), data_block
encoded encrypted IP
...
`protect end_protected
```

The manner in which the encryption tool chooses the session key is implementation-defined. It may, for example, be a default key used for all digital envelopes; however, that would be cryptographically weak. A better approach is to generate a session key randomly for use in that digital envelope only. Schemes for generation of random keys are published in the open literature and implemented in widely available software libraries.

H.3.2 Encryption for multiple users

A variation on the preceding use case allows provision of IP to multiple users' tools. The IP is encrypted using a session key and a symmetric cipher, as before, but the session key is encrypted multiple times, once for each user's tool. The encryption envelope specifies the users' keys, as follows:

```
`protect key_keyowner="ACME IP User1", key_method="rsa", key_block
`protect key_keyowner="ACME IP User2", key_method="elgamal", key_block
`protect key_keyowner="ACME IP User3", key_method="aes192-cbc", key_block
`protect data_method="aes192-cbc"
`protect begin
IP source text
...
`protect end
```

The decryption envelope generated by the encryption tool is:

```
`protect begin_protected
`protect encrypt_agent="Encryptomatic", encrypt_agent_info="2.3.4a"
`protect key_keyowner="ACME IP User1", key_method="rsa"
`protect encoding = (enctype="base64", line_length=40, bytes=256)
`protect key_block
encoded encrypted session key
...
`protect key_keyowner="ACME IP User2", key_method="elgamal"
`protect encoding = (enctype="base64", line_length=40, bytes=256)
`protect key_block
encoded encrypted session key
...
`protect key_keyowner="ACME IP User3", key_method="aes192-cbc"
`protect encoding = (enctype="base64", line_length=40, bytes=24)
`protect key_block
encoded encrypted session key
...
`protect data_method="aes192-cbc"
`protect encoding = (enctype="base64", line_length=40, bytes=4006)
`protect data_block
encoded encrypted IP
...
`protect end_protected
```

Each user's decryption tool examines the key blocks in the decryption envelope to find one encrypted using a key to which the tool has access. It then uses that key to decrypt the session key, and then uses the session key to decrypt the IP.

This example also illustrates a further variation. The cipher used to encrypt a session key need not be an asymmetric cipher. If a digital envelope is used as a means of providing IP to multiple users, the choice of cipher and key for session key encryption can be made independently for each user.

H.4 Digital signatures

A digital signature allows detection of alteration of the IP provided by an author. A scenario in which alteration might be attempted involves provision of IP to a user, encrypted with the public key of the user's tool. A malicious third party may have access to the public key, since it published by the user. The third party could spoof the IP author, for example, by intercepting the media on which IP is delivered, and provide

a substitute decryption envelope containing malicious IP. The malicious IP would also be encrypted with the public key of the user's tool. If the user were unaware of the substitution, he or she would invoke the decryption tool to decrypt the malicious IP using the tool's private key. Use of the malicious IP might cause damage to the user's business and consequential damage to the IP author.

Scenarios such as this can be avoided by having the IP author sign the IP. Signing involves application of a hash function to the IP text to compute a digest of the IP. The hash function has the property that application to different texts produces different digests. Moreover, it is not possible to reconstruct the text from a digest. The digest is encrypted using the author's private key and provided along with the IP. The only way the encrypted digest can be properly decrypted is with the author's public key, which the author has published.

The user's tool receiving the IP recomputes the digest using the same hash function on the received IP. The tool also decrypts the author's digest using the author's public key, and compares that digest with the recomputed digest. If they are the same, the user has confidence that the received IP is unaltered. If they differ, the delivery has been modified. In that case, the user should not trust the received IP.

The author includes digest directives in the encryption envelope to specify that a digital signature be used. The digest directives can specify a hash function to use and key for encrypting the digest. If either of these specifications is omitted, the encryption tool chooses the hash function or key in an implementation-defined manner. A typical choice would be to use a default hash function or a default key previously identified by the author. An example encryption envelope specifying a digital signature is:

```
`protect key_keyowner="ACME IP User", key_method="rsa", key_block
`protect data_method="aes192-cbc"
`protect digest_keyowner="ACME IP Author", digest_key_method="rsa"
`protect digest_method="sha1", digest_block
`protect begin
IP source text
...
`protect end
```

The decryption envelope produced by the tool is:

```
`protect begin_protected
`protect key_keyowner="ACME IP User", key_method="rsa"
`protect encoding = (enctype="base64", line_length=40, bytes=256), key_block
encoded encrypted session key
...
`protect data_method="aes192-cbc"
`protect encoding = (enctype="base64", line_length=40, bytes=4006), data_block
encoded encrypted IP
...
`protect digest_keyowner="ACME IP Author", digest_key_method="rsa"
`protect digest_method="sha1"
`protect encoding = (enctype="base64", line_length=40, bytes=16), digest_block
encoded encrypted digest
...
`protect end_protected
```

While this example shows a digital signature used with a digital envelope, that is not a requirement. A digital signature can augment a simple protection envelope as described in H.2.

H.5 Key exchange

Protection of IP from disclosure relies on security of encryption keys. Should a key become known to an unauthorized party, the encrypted IP can be decrypted and disseminated. In conventional encryption, the intended recipient of a message is assumed to have an interest in the security of an encrypted message and is trusted to keep keys secret. In the context of protected IP exchange, the true recipient is the user's tool, not the user. The IP author might not trust the user not to examine or use the IP in unauthorized ways. Nonetheless, the author must provide the IP to the user's tools so that the user can gain the benefit of the IP. Moreover, exchange of keys between the author and the user's tools may need to be mediated by the user. These considerations make key exchange more complicated than in many conventional applications of cryptography.

Many applications that require secure exchange of keys rely on *public key infrastructure* (PKI). Parties to communication generate, or are given, key pairs for use with asymmetric ciphers. Each party keeps their private key secret, and publishes their public key, for example, in a directory. In order to establish that a public key does, in fact, belong to a given party, the public key is digitally signed by a trusted authority. The signed public key is represented in the form of a digital certificate, containing the key and the signature. The trusted authority is called a *certification authority* (CA). Many PKI systems have a hierarchy of CAs, allowing a certificate signed by a subordinate CA to be signed by a superior CA, allowing trust to be distributed hierarchically. One or more root CAs are required to be globally trusted.

Key exchange for IP protection may be built upon public key infrastructure. For example, a vendor of a decryption tool may embed a private key of a key pair in the tool and register the public key with a CA. The tool can then generate a key pair for the tool's user, keeping the private key secret and signing the public key with both the vendor's private key and the user's private key. This allows verification that the public key originates with the instance of the vendor's tool owned by the tool user. That public key may then be used by IP authors to provide IP for that use of that tool only. Similar mechanisms might also be employed within tools to allow exchange of private keys among tools without disclosure to the tools' user.

In addition to providing for secure key exchange, a decryption tool must take measures to ensure that stored keys are not disclosed to the tool user (see 24.1.6). If a tool user could read a tool's stored keys, the user could decrypt IP independently of the tool. One way of ensuring security of a tool's keys is for the tool to encrypt its key store using a secret key embedded in the tool in a disguised manner, and to provide for update and re-encryption of the secret key in case it is compromized.

Annex I

(informative)

Glossary

For the purposes of this document, the following terms and definitions apply. These and other terms within IEEE standards are found in *The Authoritative Dictionary of IEEE Standards Terms* [B8].

This glossary contains brief, informal descriptions for a number of terms and phrases used to define this language. The complete, formal definition of each term or phrase is provided in the main body of the standard.

For each entry, the relevant clause or subclause numbers in the text are given. Some descriptions refer to multiple clauses in which the single concept is discussed; for these, the clause number containing the definition of the concept is given in italics. Other descriptions contain multiple clause numbers when they refer to multiple concepts; for these, none of the clause numbers are italicized.

absolute design hierarchy search string: A search string provided to the `vhpi_handle_by_name` function that represents the full name of an object in the design hierarchy information model. (23.2)

absolute library search string: A search string provided to the `vhpi_handle_by_name` function that represents the full name of an object in the library information model. (23.2)

abstract class: A class that cannot be the most specialized class of any object. (17.2.1)

abstract literal: A literal of the *universal_real* abstract type or the *universal_integer* abstract type. (15.3, *15.5.1*)

access mode: The mode in which a file object is opened, which can be either *read-only* or *write-only*. The access mode depends on the value supplied to the Open_Kind parameter. (*5.5.2*, 16.3)

access type: A type that provides access to an object of a given type. Access to such an object is achieved by an access value returned by an allocator; the access value is said to *designate* the object. (*5.1, 5.4*)

access value: A value of an access type. This value is returned by an allocator and designates an object (which shall be a variable) of a given type. A null access value designates no object. An access value can only designate an object created by an allocator; it cannot designate an object declared by an object declaration. (*5.1, 5.4*)

action callback: A callback whose trigger event relates to occurrence of phases of tool execution and other aspects of tool execution. (21.3.7)

active driver: A driver that acquires a new value during a simulation cycle regardless of whether the new value is different from the previous value. (14.7.3.1, 14.7.5)

actual: An expression, a port, a signal, a variable, a subtype, a subprogram, or a package associated with a formal port, formal parameter, or formal generic. (5.3.2.2, *6.4.2.3*, *6.5.6.2, 6.5.6.3,* 6.5.7.1*, 6.5.7.2*, *6.5.7.3*, 7.3.2)

aggregate: (A) The kind of expression, denoting a value of a composite type. The value is specified by giving the value of each of the elements of the composite type. Either a positional association or a named association shall be used to indicate which value is associated with which element. (**B**) A kind of target of a variable assignment statement or signal assignment statement assigning a composite value. The target is then said to *be in the form of an aggregate*. (9.3.2, *9.3.3*, 9.3.5, 9.3.6, 9.4.3)

alias: An alternate name for a named entity. (6.6)

allocator: An operation used to create anonymous, variable objects accessible by means of access values. (*5.4*, 9.3.7)

analysis: The syntactic and semantic analysis of source code in a VHDL design file and the insertion of intermediate form representations of design units into a design library. (13.1, 13.2, 13.5)

analysis phase: That phase of tool execution in which analysis of a design file occurs. (Clause 13, *20.3*)

anonymous: The undefined simple name of an item, which is created implicitly. The base type of a numeric type or an array type is anonymous; similarly, the object denoted by an access value is anonymous. (6.2)

application context: The application context of a class specifies whether objects of the class may exist in either or both of the library information model or the design hierarchy information model, and as a consequence, when the object is accessible to VHPI programs. (19.8)

application name: An identifier that, jointly with an object library name, uniquely identifies a foreign application. (20.2.2)

appropriate: A prefix is said to be appropriate for a type if the type of the prefix is the type considered, or if the type of the prefix is an access type whose designated type is the type considered. (8.1)

architecture body: A body associated with an entity declaration to describe the internal organization or operation of a design entity. An architecture body is used to describe the behavior, dataflow, or structure of a design entity. (Clause 3, *3.3*)

array object: An object of an array type. (*Clause 5*)

array type: A type, the value of which consists of elements that are all of the same subtype (and hence, of the same type). Each element is uniquely distinguished by an index (for a one-dimensional array) or by a sequence of indexes (for a multidimensional array). Each index shall be a value of a discrete type and shall lie in the correct index range. (5.3.2)

ascending range: A range L to R. (5.2.1)

ASCII: American Standard Code for Information Interchange. The package Standard contains the definition of the type CHARACTER, the first 128 values of which represent the ASCII character set. (5.2.2.2, *16.3*)

assertion violation: A violation that occurs when the condition of an assertion statement evaluates to false. (10.3)

associated driver: The single driver for a signal in the (explicit or equivalent) process statement containing the signal assignment statement. (14.7.2)

associated individually: A property of a formal port, generic constant, or parameter of a composite type with respect to some association list. A composite formal whose association is defined by multiple association elements in a single association list is said to be *associated individually* in that list. The formats of such association elements shall denote non-overlapping subelements or slices of the formal. (6.5.7.1)

associated in whole: When a single association element of a composite formal supplies the association for the entire formal. (6.5.7.1)

association element: An element that associates an actual or local with a local or formal. (6.5.7.1)

association list: A list that establishes correspondences between formal or local port or parameter names and local or actual names or expressions. (6.5.7.1)

association relationship: A relationship between objects in an information model that has semantic significance. (17.2.1)

asymmetric cipher: A cipher requiring one key of a key pair for encryption of information and the other key of the pair for decryption. The owner of the key pair keeps one key of the pair private (the *private key*) and publishes the other key (the *public key*). (24.1.1, 24.1.3.2)

attribute: A definition of some characteristic of a named entity. Some attributes are predefined for types, ranges, values, signals, and functions. The remaining attributes are user defined and are always constants. (6.7)

based literal: An abstract literal expressed in a form that specifies the base explicitly. The base is restricted to the range 2 to 16. (15.5.3)

base specifier: A lexical element that indicates whether a bit string literal is to be interpreted as a binary, octal, decimal, or hexadecimal value. (15.8)

base type: The type from which a subtype defines a subset of possible values, otherwise known as a *constraint*. This subset is not required to be proper. The base type of a type is the type itself. The base type of a subtype is found by recursively examining the type mark in the subtype indication defining the subtype. If the type mark denotes a type, that type is the base type of the subtype; otherwise, the type mark is a subtype, and this procedure is repeated on that subtype. (5.1) *See also:* **subtype**.

basic operation: An operation that is inherent in one of the following:

- An assignment (in an assignment statement or initialization)
- An allocator
- A selected name, an indexed name, or a slice name
- A qualification (in a qualified expression), an explicit type conversion, a formal or actual designator in the form of a type conversion, or an implicit type conversion of a value of type *universal_integer* or *universal_real* to the corresponding value of another numeric type, or
- A numeric literal (for a universal type), the literal null (for an access type), a string literal, a bit string literal, an aggregate, or a predefined attribute (5.1)

basic signal: A signal that determines the driving values for all other signals. A basic signal is

- Either a scalar signal or a resolved signal
- Not a subelement of a resolved signal
- Not an implicit signal of the form S'STABLE(T), S'QUIET(T), or S'TRANSACTION, and
- Not an implicit signal GUARD (14.7.3.2)

belong: (**A**) (**to a range**): A property of a value with respect to some range. The value V is said to *belong to a range* if the relations (lower bound <= V) and (V <= upper bound) are both true, where lower bound and upper bound are the lower and upper bounds, respectively, of the range. (*5.2.1*, 5.3.2) (**B**) (**to a subtype**): A property of a value with respect to some subtype. A value is said to *belong to a subtype* of a given type if it belongs to the type and satisfies the applicable constraint. (5.1, 5.3.2)

binding: The process of associating a design entity and, optionally, an architecture with an instance of a component. A binding can be specified in an explicit or a default binding indication. (3.4, *7.3.2, 7.3.3*, 14.4.3.3, 14.5.4)

bit string literal: A literal formed by a sequence of extended digits enclosed between two quotation (") characters and preceded by a base specifier. The type of a bit string literal is determined from the context. (9.3.2, *15.8*)

block: (**A**) The representation of a portion of the hierarchy of a design. A block is either an external block or an internal block. (*3.1*, 3.3.2, 3.4.1, *3.4.2*, 3.4.3, 6.5.6.2, 6.5.6.3) (**B**) The act of suspending the execution of a process for the purposes of guaranteeing exclusive access to either a file object or an object of a protected type. (5.5.2, 14.6)

bound: A label that is identified in the instantiation list of a configuration specification. (7.3.1)

box: (A) The symbol <> in an index subtype definition, which stands for an undefined range. Different objects of the type need not have the same bounds and direction. (5.3.2.1) **(B)** The symbol <> as the subprogram default in a formal generic subprogram declaration, and which stands for a subprogram with the same name and parameter and result type profile as the formal subprogram visible at the place of instantiation of the enclosing uninstantiated declaration. (6.5.4) **(C)** The symbol <> in an interface package generic map aspect indicating that the actual instantiated package may have any actual generics. (6.5.5)

buffer: One possible port mode. A port of mode **buffer** contributes its driving value to the network containing the port; the design entity containing the port is also allowed to read its driving value. (6.5.2, 6.5.6.3)

bus: One kind of guarded signal. A bus floats to a user-specified value when all of its drivers are turned off. (6.4.2.3, 6.5.2)

callback: A mechanism for a VHPI program to gain control during tool execution. (Clause 21)

callback data structure: A C struct of type `vhpiCbDataT` that specifies a callback. It is used to register a callback and to acquire information about a callback and is passed to a callback function upon invocation of the function. (21.2.2, 21.2.5, 21.2.6)

callback function: A function in a VHPI program, identified to the tool by registration, that is called by the tool upon occurrence of a nominated trigger event. (21.1)

callback reason: A specification of an occurrence that may trigger invocation of a callback function. (21.1)

capability set: A permissible subset of the VHPI information model and functions provided by a tool. (17.3)

change: The current value of a signal of type T is said to change as the result of an update if and only if application of the predefined "=" operator for type T to the current value of the signal and the value of the signal prior to the update evaluates to FALSE. (14.7.3.4)

character literal: A literal of the CHARACTER type. Character literals are formed by enclosing one of the graphic characters (including the space and nonbreaking space characters) between two apostrophe (') characters. (15.3, *15.6*)

character type: An enumeration type with at least one of its enumeration literals as a character literal. (*5.2.2, 5.2.2.2*)

chosen implementation: An implementation of floating-point types that conforms to either IEEE Std 754-1985 or to IEEE Std 854-1987 and with a minimum representation size of 64 bits. (5.2.5.1)

cipher: An algorithm for encrypting and decrypting information. A cipher is either symmetric, requiring a single secret key for both encryption and decryption, or asymmetric, requiring one key of a key pair for encryption and the other key of the pair for decryption. (24.1.1, 24.1.3.2)

class: An abstract data type within an information model. (17.2.1)

closely related types: Two type marks that denote the same type or two numeric types. Two array types are closely related if they have the same dimensionality and if the element types are closely related. Explicit type conversion is only allowed between closely related types. (9.3.6)

comment: Informative text added to a description. (15.9)

complete: A loop that has finished executing. Similarly, an iteration scheme of a loop is complete when the condition of a while iteration scheme is FALSE or all of the values of the discrete range of a for iteration scheme have been assigned to the iteration parameter. (10.10)

complete context: A declaration, a specification, a statement, or, in certain cases, a discrete range or an expression; complete contexts are used in overload resolution. (12.5)

composite type: A type whose values have elements. There are two classes of composite types: *array types* and *record types*. (5.1, *5.3*)

concurrent region: A block declarative region (including an external block and any block equivalent to a generate statement), or a package declarative region (including a generic-mapped package equivalent to a package instantiation) declared immediately within a concurrent region. (8.7)

concurrent statement: A statement that executes asynchronously, with no defined relative order. Concurrent statements are used for dataflow and structural descriptions. (Clause 11)

configuration: A construct that defines how component instances in a given block are bound to design entities in order to describe how design entities are put together to form a complete design. (*3.1*, 3.4. 7.3)

conforming profiles: Two subprogram declarations are said to have conforming profiles if and only if both are procedures or both are functions, the parameter and result type profiles of the subprograms are the same and, at each parameter position, the corresponding parameters have the same class and mode. (4.10)

connected: A formal port associated with an actual port or signal. A formal port associated with the reserved word open is said to be *unconnected*. (6.5.6.3)

constant: An object whose value cannot be changed. Constants are either *explicitly declared*, subelements of explicitly declared constants, or interface constants. Constants declared in packages can also be *deferred constants*. (6.4.2.2)

constraint: A subset of the values of a type. The set of possible values for an object of a given type that can be subjected to a condition is called a *constraint*. A value is said to *satisfy* the constraint if it satisfies the corresponding condition. There are index constraints and range constraints. (5.1)

contributor: A contributor of a given signal is a driver, signal, expression, or conversion whose value determines the value of the given signal. (19.12.2)

conversion function: A function used to convert values flowing through associations. For interface objects of mode **in**, conversion functions are allowed only on actuals. For interface objects of mode **out** or **buffer**, conversion functions are allowed only on formals. For interface objects of mode **inout** or **linkage**, conversion functions are allowed on both formals and actuals. Conversion functions have a single parameter. A conversion function associated with an actual accepts the type of the actual and returns the type of the formal. A conversion function associated with a formal accepts the type of the formal and returns the type of the actual. (6.5.7.1)

convertible: A property of an operand with respect to some type. An operand is convertible to some type if there exists an implicit conversion to that type. (9.3.6)

current value: The value component of the single transaction of a driver whose time component is not greater than the current simulation time. (14.4.1, *14.7.2*, 14.7.3. 14.7.4)

decimal literal: An abstract literal that is expressed in decimal notation. The base of the literal is implicitly 10. The literal may optionally contain an exponent or a decimal point and fractional part. (15.5.2)

declaration: A construct that defines a declared entity and associates an identifier (or some other notation) with it. This association is in effect within a region of text that is called the *scope* of the declaration. Within the scope of a declaration, there are places where it is possible to use the identifier to refer to the associated declared entity; at such places, the identifier is said to be the *simple name* of the named entity. The simple name is said to *denote* the associated named entity. (Clause 6)

declarative part: A syntactic component of certain declarations or statements (such as entity declarations, architecture bodies, and block statements). The declarative part defines the lexical area (usually introduced by a reserved word such as **is** and terminated with another reserved word such as **begin**) within which declarations may occur. (*3.2.3, 3.3.2*, 3.4.1, 4.8, 11.2, 11.3, 11.7.2, 11.7.3)

declarative region: A semantic component of certain declarations or statements. Certain declarative regions include disjoint parts; for example, the declarative region of a package declaration, which, if there is an associated package body, extends to the end of that package body. (12.1)

decorate: To associate a user-defined attribute with a named entity and to define the value of that attribute. (7.2)

decryption envelope: A collection of protect tool directives that specify ciphers and keys used to decrypt a portion of a VHDL description. The decryption envelope also contains the encoded encrypted portion of the VHDL description. (24.1.5)

decryption tool: A tool that processes decryption envelopes in a VHDL description to yield the original source text. The decryption tool may perform subsequent analysis and interpretation of the description, but shall not disclose the decrypted text to the user of the tool. (24.1.5)

default expression: A default value that is used for a formal generic constant, port, or parameter if the interface object is unassociated. A default expression is also used to provide an initial value for signals and their drivers. (*6.4.2.3, 6.5.7*)

deferred constant: A constant that is declared without an assignment symbol (:=) and expression in a package declaration. A corresponding full declaration of the constant shall exist in the package body to define the value of the constant. (6.4.2.2)

delimited comment: A comment that starts with a solidus (slash) character immediately followed by an asterisk character and extends up to the first subsequent occurrence of an asterisk character immediately followed by a solidus character. (15.9)

delta cycle: A simulation cycle in which the simulation time at the beginning of the cycle is the same as at the end of the cycle. That is, simulation time is not advanced in a delta cycle. Only nonpostponed processes can be executed during a delta cycle. (14.7.5.1)

denote: A property of the identifier given in a declaration. Where the declaration is visible, the identifier given in the declaration is said to *denote* the named entity declared in the declaration. (6.1)

depend: (A) (on a library unit): A design unit that explicitly or implicitly mentions other library units in a use clause. These dependencies affect the allowed order of analysis of design units. (13.5) (**B**) (**on a signal value**): A property of a signal with respect to some other signal. The current value of an implicit signal R is said to *depend on* the current value of another signal S if R denotes an implicit signal S'STABLE(T), S'QUIET(T), or S'TRANSACTION, or if R denotes an implicit GUARD signal and S is any other implicit signal named within the guard condition that defines the current value of R. The current value of an interface signal R is said to depend on the current value of an implicit signal S if R denotes a port of mode in and S is the actual associated with that port. (14.7.4)

deposit: An update of the current value of a variable other than by assignment, of a driver other than by advancement of a transaction to the first position in the driver's projected output waveform, or of a signal other than resulting from update of other parts of the net of which the signal is a part. A deposited value remains only until a subsequent update of the variable, driver, or signal. (14.7.2, 14.7.3, 22.5.2, 22.5.3, 22.5.4)

descending range: A range L **downto** R. (5.2.1)

design entity: An entity declaration together with an associated architecture body. Different design entities may share the same entity declaration, thus describing different components with the same interface or different views of the same component. (3.1)

design file: One or more design units in sequence. (13.1)

design hierarchy: The complete representation of a design that results from the successive decomposition of a design entity into subcomponents and binding of those components to other design entities that may be decomposed in a similar manner. (3.1)

design hierarchy information model: The information model that represents the elaborated VHDL model. (17.2.1)

design library: A host-dependent storage facility for intermediate-form representations of analyzed design units. (13.2)

design unit: A construct that can be independently analyzed and stored in a design library. A design unit is either an entity declaration, an architecture body, a configuration declaration, a package declaration, a package body, a package instantiation declaration, a context declaration, or a PSL verification unit. (13.1)

designate: A property of access values that relates the value to some object when the access value is non-null. A non-null access value is said to *designate* an object. (5.4.1)

designated subtype: For an access type, the subtype defined by the subtype indication of the access type definition. (5.4.1)

designated type: For an access type, the base type of the subtype defined by the subtype indication of the access type definition. (5.4.1)

designator: (**A**) Syntax that forms part of an association element. A formal designator specifies which formal parameter, port, or generic (or which subelement or slice of a parameter, port, or generic) is to be associated with an actual by the given association element. An actual designator specifies which actual expression, signal, variable, subtype, subprogram, or package is to be associated with a formal (or subelement or subelements of a formal). An actual designator may also specify that the formal in the given association element is to be left unassociated (with an actual designator of **open**). (6.5.7.1) (**B**) An identifier, character literal, or operator symbol that defines an alias for some other name. (6.6.1) (**C**) A simple name that denotes a predefined or user-defined attribute in an attribute name, or a user-defined attribute in an attribute specification. (7.2, 8.6) (**D**) A simple name, character literal, or operator symbol, and possibly a signature, that denotes a named entity in the entity name list of an attribute specification. (7.2) (**E**) An identifier or operator symbol that defines the name of a subprogram. (4.2.1) (**F**) An identifier, character literal, or operator symbol associated with a named entity by a declaration. (6.1)

digest: A summary of information, computed using a hash function. (24.1.1)

digital envelope: An encryption scheme in which information is encrypted using a symmetric cipher with a session key chosen by an encryption tool, and then the session key is encrypted. Decryption of the protected envelope involves first decrypting the session key, followed by decrypting the information with the symmetric cipher using the decrypted session key. (24.1.1)

digital signature: A scheme that allows verification that information is received unaltered from the originator of the information. The originator computes a digest of the information using a hash function and encrypts the digest with an asymmetric cipher using the private key of a key pair. The recipient decrypts the digest using the public key of the originator, recomputes the digest by applying the hash function to the received information, and compares the two digests. If they differ, the received information differs from the originator's information. (24.1.1)

directly visible: A visible declaration that is not visible by selection. A declaration is directly visible within its immediate scope, excluding any places where the declaration is hidden. A declaration occurring immediately within the visible part of a package can be made directly visible by means of a use clause. (*12.3*, 12.4) *See also:* **visible**.

disabled callback: A callback for which the callback function will not be called if the trigger event occurs. (21.1)

discrete array: A one-dimensional array whose elements are of a discrete type. (9.2.3)

discrete range: A range whose bounds are of a discrete type. (5.3.2.1, 5.3.2.2)

discrete type: An enumeration type or an integer type. Each value of a discrete type has a position number that is an integer value. Indexing and iteration rules use values of discrete types. (5.2.1)

don't care value: The enumeration literal '–' of the type STD_ULOGIC defined in the package STD_LOGIC_1164. (16.8.2.2)

driver: A container for a projected output waveform of a signal. The value of the signal is a function of the current values of its drivers. Each process that assigns to a given signal implicitly contains a driver for that signal. A signal assignment statement affects only the associated driver(s). (14.5.5, *14.7.2*, 14.7.3, 14.7.4)

driving value: The value a signal provides as a source of other signals. (14.7.3)

driving-value forced signal: A signal whose driving value is set to a given value and cannot be changed by a deposit or update of other parts of the net of which the signal is a part. (14.7.3.2, *22.5.3*)

dynamic object: An object in an information model that, once created, may cease to exist at a later time during execution of the tool. (17.2.1)

effective value: The value obtained by evaluating a reference to the signal within an expression. (14.7.3)

effective-value forced signal: A signal whose effective value is set to a given value and cannot be changed by a deposit or update of other parts of the net of which the signal is a part. (14.7.3.3, *22.5.3*)

elaboration: The process by which a declaration achieves its effect. Prior to the completion of its elaboration (including before the elaboration), a declaration is not yet elaborated. (Clause 14)

elaboration function: A function in a foreign architecture that performs elaboration of the foreign architecture. (20.4.1)

elaboration phase: That phase of tool execution in which static elaboration of a design hierarchy occurs. (14.2, *20.4*)

element: A constituent of a composite type. (5.1) *See also:* **subelement**.

enabled callback: A callback for which the callback function will be called if the trigger event occurs. (21.1)

encoding method: An algorithm that transforms the octets of information into graphic characters so that the information can be stored or transmitted without being altered by agents that interpret non-graphic characters. (24.1.1, 24.1.3.1)

encryption envelope: A collection of protect tool directives that specify ciphers and keys used to encrypt an enclosed portion of a VHDL description. (24.1.4)

encryption tool: A tool that processes encryption envelopes in a VHDL description and produces a VHDL description containing the corresponding decryption envelopes. (24.1.4)

entity declaration: A definition of the interface between a given design entity and the environment in which it is used. It may also specify declarations and statements that are part of the design entity. A given entity declaration may be shared by many design entities, each of which has a different architecture. Thus, an entity declaration can potentially represent a class of design entities, each with the same interface. (*3.1*, 3.2)

enumeration literal: A literal of an enumeration type. An enumeration literal is either an identifier or a character literal. (*5.2.2.1*, 9.3.2)

enumeration type: A type whose values are defined by listing (enumerating) them. The values of the type are represented by enumeration literals. (*5.2.1*, 5.2.2)

erroneous: An error condition that cannot always be detected. (1.3.3)

error: A condition that makes the source description illegal. If an error is detected at the time of analysis of a design unit, it prevents the creation of a library unit for the given design unit. A runtime error causes simulation to terminate. (1.3.3, 13.5)

error information structure: A C struct of type `vhpiErrorInfoT` that represents error information provided by the tool to a VHPI program upon occurrence of an error. (23.3)

event: A change in the current value of a signal, which occurs when the signal is updated with its effective value. (14.7.3.4)

execute: (A) When first the design hierarchy of a model is elaborated, then its nets are initialized, and finally simulation proceeds with repetitive execution of the simulation cycle, during which processes are executed and nets are updated. **(B)** When a process performs the actions specified by the algorithm described in its statement part. (Clause 14, 14.7)

execution function: A function in a foreign model that performs initialization (in the case of a foreign architecture) or dynamic elaboration (in the case of a foreign subprogram). (20.4.2, 20.5)

expanded name: A selected name (in the syntactic sense) that denotes one or all of the primary units in a library or any named entity within a primary unit. (*8.3*, 10.2) *See also:* **selected name**.

explicit ancestor: The parent of the implicit signal that is defined by the predefined attributes 'DELAYED, 'QUIET, 'STABLE, or 'TRANSACTION. It is determined using the prefix of the attribute. If the prefix denotes an explicit signal or a slice or subelement (or member thereof), then that is the explicit ancestor of the implicit signal. If the prefix is one of the implicit signals defined by the predefined attributes 'DELAYED, 'QUIET, 'STABLE, or 'TRANSACTION, this rule is applied recursively. If the prefix is an implicit signal GUARD, the signal has no explicit ancestor. (4.3)

explicit signal: A signal, other than those defined by the predefined attributes 'DELAYED, 'QUIET, 'STABLE, or 'TRANSACTION, any implicit signal GUARD, or their slices, subelements, or slices of their subelements. A slice, subelement, or a slice of a subelement of an explicit signal is also an explicit signal. (4.3)

explicitly declared constant: A constant of a specified type that is declared by a constant declaration. (6.4.2.2)

explicitly declared object: An object of a specified type that is declared by an object declaration. An object declaration is called a *single-object declaration* if its identifier list has a single identifier; it is called a *multiple-object declaration* if the identifier list has two or more identifiers. (6.4.1, *6.4.2*) *See also:* **implicitly declared object**.

expression: A formula that defines the computation of a value. (9.1)

extend: A property of source text forming a declarative region with disjoint parts. In a declarative region with disjoint parts, if a portion of text is said to *extend* from some specific point of a declarative region to the end of the region, then this portion is the corresponding subset of the declarative region (and does not include intermediate declarative items between an interface declaration and a corresponding body declaration). (12.1)

extended digit: A lexical element that is either a digit or a letter. (15.4.3)

external block: A top-level design entity that resides in a library and may be used as a component in other designs. (3.1)

file type: A type that provides access to objects containing a sequence of values of a given type. File types are typically used to access files in the host system environment. The value of a file object is the sequence of values contained in the host system file. (5.1, *5.5*)

floating-point types: A scalar type whose values approximate real numbers. The representation of a floating-point type conforms either to IEEE Std 754-1985 or to IEEE Std 854-1987 and has a minimum size of 64 bits. (5.2.1, *5.2.5*)

forced driver: A driver whose current value is set to a given value and cannot be changed by a deposit or a transaction becoming the first transaction in the driver's projected output waveform. (14.7.2, *22.5.4*)

forced variable: A variable whose value is set to a given value and cannot be changed by a deposit or assignment. (10.6.2.1, *22.5.2*)

foreign application: A VHPI program other than a foreign model. (20.1)

foreign model: A design entity whose architecture is decorated with the 'FOREIGN attribute in the form of a standard indirect binding or a standard direct binding, or a subprogram similarly decorated. (20.1)

foreign model callback: A callback that allows a foreign model to achieve an effect similar to that of a wait statement, by being triggered after a timeout or upon an event on one or more signals. (21.3.3)

foreign subprogram: A subprogram that is decorated with the attribute 'FOREIGN, defined in package STANDARD. The STRING value of the attribute may specify implementation-dependent information about the foreign subprogram. Foreign subprograms may have non-VHDL implementations. An implementation may place restrictions on the allowable modes, classes, and types of the formal parameters to a foreign subprogram, such as constraints on the number and allowable order of the parameters. (4.3)

formal: A formal port or formal generic of a design entity, a block statement, or a formal parameter of a subprogram. (*4.2.2*, *6.5.7.1*, *6.5.7.2*, *6.5.7.3*, 11.2)

format: The format of a value structure specifies how the value is represented. (22.2.8)

full declaration: A constant declaration occurring in a package body with the same identifier as that of a deferred constant declaration in the corresponding package declaration. A full type declaration is a type declaration corresponding to an incomplete type declaration. (4.8)

fully bound: A binding indication for the component instance implies an entity declaration and an architecture. (7.3.2.2)

generate parameter: A constant object whose type is the base type of the discrete range of a generate parameter specification. A generate parameter is declared by a generate statement. (11.8)

generic: An interface declaration in the block header of a block statement, a component declaration, or an entity declaration, in the package header of a package declaration, or in the subprogram header of a subprogram specification. Generics provide a channel for static information to be communicated to a block, a package, or a subprogram from its environment. Unlike explicit declarations, however, the value or subtype denoted by a generic can be supplied externally, either in a component instantiation statement, in a configuration specification, or in a package or subprogram instantiation declaration. (6.5.6.2)

generic interface list: A list that defines local or formal generics. (6.5.6.1, 6.5.6.2)

generic-mapped package: A package declared by a package declaration containing a generic clause and a generic map aspect. A generic-mapped package may be declared explicitly or may be equivalent to a package instantiation. (4.7, 4.9)

generic-mapped subprogram: A subprogram declared by a subprogram declaration containing a generic list and a generic map aspect. A generic-mapped subprogram may be declared explicitly or may be equivalent to a subprogram instantiation. (4.2.1, 4.4)

globally static expression: An expression that can be evaluated as soon as the design hierarchy in which it appears is elaborated. A locally static expression is also globally static unless the expression appears in a dynamically elaborated context. (9.4.1)

globally static primary: A primary whose value can be determined during the elaboration of its complete context and that does not thereafter change. Globally static primaries can only appear within statically elaborated contexts. (9.4.3)

group: A named collection of named entities. Groups relate different named entities for the purposes not specified by the language. In particular, groups may be decorated with attributes. (6.9, 6.10)

guard: *See:* **guard condition**.

guard condition: A Boolean-valued expression associated with a block statement that controls assignments to guarded signals within the block. A guard condition defines an implicit signal GUARD that may be used to control the operation of certain statements within the block. (*6.4.2.3*, 11.2, 11.6)

guarded assignment: A concurrent signal assignment statement that includes the reserved word **guarded**, which specifies that the signal assignment statement is executed when a signal GUARD changes from FALSE to TRUE, or when that signal has been TRUE and an event occurs on one of the signals referenced in the corresponding GUARD condition. The signal GUARD shall be one of the implicitly declared GUARD signals associated with block statements that have guard conditions, or it shall be an explicitly declared signal of type BOOLEAN that is visible at the point of the concurrent signal assignment statement. (11.6)

guarded signal: A signal declared as a register or a bus. Such signals have special semantics when their drivers are updated from within guarded signal assignment statements. (6.4.2.3)

guarded target: A signal assignment target consisting only of guarded signals. An unguarded target is a target consisting only of unguarded signals. (11.6)

handle: An opaque reference to an object in the VHPI information model. (17.4.1)

hash function: A function that produces a summary of information. The likelihood of two different pieces of information yielding the same summary is negligible. Moreover, the original information cannot be determined from the summary. (24.1.1, 24.1.3.3)

hidden: A declaration that is not directly visible. A declaration is *hidden* in its scope by a homograph of the declaration. (12.3)

high-impedance value: The enumeration literal ‘Z’ of the type STD_ULOGIC defined in the package STD_LOGIC_1164. (16.8.2.2)

homograph: A reflexive property of two declarations. Each of two declarations is said to be a *homograph* of the other if both declarations have the same identifier and overloading is allowed for at most one of the two. If overloading is allowed for both declarations, then each of the two is a homograph of the other if they have the same identifier, operator symbol, or character literal, as well as the same parameter and result type profile. (3.4.2, *12.3*)

identify: A property of a name appearing in an element association of an assignment target in the form of an aggregate. The name is said to *identify* a signal or variable and any subelements of that signal or variable. (10.5.2.1, 10.6.2.1)

immediate scope: A property of a declaration with respect to the declarative region within which the declaration immediately occurs. The immediate scope of the declaration extends from the beginning of the declaration to the end of the declarative region. (12.2)

immediately within: A property of a declaration with respect to some declarative region. A declaration is said to occur *immediately within* a declarative region if this region is the innermost region that encloses the declaration, not counting the declarative region (if any) associated with the declaration itself. (12.1)

implicit label: Where a statement omits a label, an implicit label is used to construct name properties for the statement. The implicit label is determined by the statement's position in the immediately enclosing statement part. (19.4.2)

implicitly declared object: An object whose declaration is not explicit in the source description, but is a consequence of other constructs; for example, signal GUARD. (*6.4.1*, 11.2, 16.2) *See also:* **explicitly declared object**.

implicit signal: Any signal S'STABLE(T), S'QUIET(T), S'DELAYED(T), or S'TRANSACTION, or any implicit GUARD signal. A slice or subelement (or slice thereof) of an implicit signal is also an implicit signal. (14.7.3, 14.7.4, 14.7.5)

imply: A property of a binding indication in a configuration specification with respect to the design entity indicated by the binding indication. The binding indication is said to *imply* the design entity; the design entity is indicated directly, indirectly, or by default. (7.3.2.2)

impure function: A function that may return a different value each time it is called, even when different calls have the same actual parameter values. A pure function returns the same value each time it is called using the same values as actual parameters. An impure function can update objects outside of its scope and can access a broader class of values than a pure function. (4.1)

in: One possible mode of a port or subprogram parameter; also, the only allowed mode of a generic constant. A port of mode **in** may be read within the design entity containing the port but does not contribute a driving value to the network containing the port. A subprogram parameter of mode **in** may be read but not modified by the containing subprogram. (4.2.2, 6.5.2, 6.5.6.2, 6.5.6.3)

incomplete type declaration: A type declaration that is used to define mutually dependent and recursive access types. (5.4.2)

incremental binding: A binding indication in a configuration declaration that either reassociates a previously associated local generic constant or that associates a previously unassociated local port is said to *incrementally rebind* the component instance or instances to which the binding indication applies. (7.3.2.1)

index constraint: A constraint that determines the index range for every index of an array type, and thereby the bounds of the array. An index constraint is *compatible* with an array type if and only if the constraint defined by each discrete range in the index constraint is compatible with the corresponding index subtype in the array type. An array value *satisfies* an index constraint if the array value and the index constraint have the same index range at each index position. (5.2.1, *5.3.2.2*)

index range: A multidimensional array has a distinct element for each possible sequence of index values that can be formed by selecting one value for each index (in the given order). The possible values for a given

index are all the values that belong to the corresponding range. This range of values is called the *index range*. (5.3.2.1)

index subtype: For a given index position of an array, the *index subtype* is denoted by the type mark of the corresponding index subtype definition. (5.3.2.1)

inertial delay: A delay model used for switching circuits; a pulse whose duration is shorter than the switching time of the circuit will not be transmitted. Inertial delay is the default delay mode for signal assignment statements. (10.5.2.1) *See also:* **transport delay**.

information model: An abstract representation of the topology and state of a VHDL model. (17.2.1)

inheritance relationship: A relationship between a subclass and a superclass whereby the subclass implicitly has all of the properties, operations, and associations of the superclass. The relationship may be directly between a subclass and a superclass or indirectly through one or more intermediate superclasses. (17.2.1)

initial value expression: An expression that specifies the initial value to be assigned to a variable. (6.4.2.4)

initialization phase: That phase of tool execution in which initialization of an elaborated design hierarchy occurs. (14.7.5.2, *20.5*)

inout: One possible mode of a port or subprogram parameter. A port of mode **inout** may be read within the design entity containing the port and also contributes a driving value to the network containing the port. A subprogram parameter of mode **inout** may be both read and modified by the containing subprogram. (4.2.2.1, 6.5.2, 6.5.6.3)

inputs: The signals identified by the longest static prefix of each signal name appearing as a primary in each expression (other than time expressions) within a concurrent signal assignment statement. (11.6)

instance: A subcomponent of a design entity whose prototype is a component declaration, design entity, or configuration declaration. Each instance of a component may have different actuals associated with its local ports and generics. A component instantiation statement whose instantiated unit denotes a component creates an instance of the corresponding component. A component instantiation statement whose instantiated unit denotes either a design entity or a configuration declaration creates an instance of the denoted design entity. (11.7.1, 11.7.2, 11.7.3)

integer literal: An abstract literal of the type *universal_integer* that does not contain a base point. (15.5.1)

integer type: A discrete scalar type whose values represent integer numbers within a specified range. (*5.2.1*, 5.2.3)

interface list: A list that declares the interface objects required by a subprogram, component, design entity, or block statement. (6.5.6)

internal block: A nested block in a design unit, as defined by a block statement. (3.1)

invalid handle: A handle that previously referred to an object that subsequently ceased to exist. (17.4.5)

ISO: International Organization for Standardization.

ISO/IEC 8859-1: The ISO Latin-1 character set. Package STANDARD contains the definition of type CHARACTER, which represents the ISO Latin-1 character set. (5.2.2.2, 16.3)

kernel process: A conceptual representation of the agent that coordinates the activity of user-defined processes during a simulation. The kernel process causes the execution of I/O operations, the propagation of signal values, and the updating of values of implicit signals [such as S'STABLE(T)]; in addition, it detects events that occur and causes the appropriate processes to execute in response to those events. (14.7.1)

left bound: For a range L **to** R or L **downto** R, the value L. (5.2.1)

left of: When both a value V1 and a value V2 belong to a range and either the range is an ascending range and V2 is the successor of V1, or the range is a descending range and V2 is the predecessor of V1. (5.2.1)

left-to-right order: When each value in a list of values is to the left of the next value in the list within that range, except for the last value in the list. (5.2.1)

lexically conform: Two subprogram specifications are said to lexically conform if, apart from certain allowed minor variations, both specifications are formed by the same sequence of lexical elements, and corresponding lexical elements are given the same meaning by the visibility rules. Lexical conformance is defined similarly for deferred constant declarations. (4.10)

library: *See:* **design library**.

library information model: The information model that represents the design units that comprise a VHDL model after analysis and prior to elaboration. (17.2.1)

library unit: The representation in a design library of an analyzed design unit. (13.1)

lifetime of an object: The duration of existence of the object in the VHPI information model. (17.4.5)

linkage: One possible port mode. A design entity whose entity interface contains a port of mode **linkage** implies that the behavior of the design entity is not expressed in terms of VHDL semantics. (6.5.2, 6.5.6.3)

literal: A value that is directly specified in the description of a design. A literal can be a bit string literal, enumeration literal, numeric literal, string literal, or the literal **null**. (9.3.2)

load: A load of a given signal is a process, port, signal, or conversion whose value depends on the value of the given signal. (19.12.2)

local contributor: A contributor defined by a VHDL model or created using the `vhpi_create` function, prior to any optimization of the representation of contributors and loads of a net. (19.12.2)

local generic: An interface declaration in a component declaration that serves to connect a formal generic in the interface list of an entity and an actual generic, value, subtype, subprogram, or package in the design unit instantiating that entity. (6.4.1, 6.5.7, 6.8)

local load: A load defined by a VHDL model, prior to any optimization of the representation of contributors and loads of a net. (19.12.2)

locally static expression: An expression that can be evaluated during the analysis of the design unit in which it appears. (9.4.1, 9.4.2)

locally static name: A name in which every expression is locally static (if every discrete range that appears as part of the name denotes a locally static range or subtype and if no prefix within the name is either an object or value of an access type or a function call). (8.1)

locally static primary: One of a certain group of primaries that includes literals, certain constants, and certain attributes. (9.4.2)

locally static subtype: A subtype whose bounds and direction can be determined during the analysis of the design unit in which it appears. (9.4.2)

local port: A signal declared in the interface list of a component declaration that serves to connect a formal port in the interface list of an entity and an actual port or signal in the design unit instantiating that entity. (6.4.1, 6.5.7, 6.8)

longest static prefix: The name of a signal or a variable name, if the name is a static signal or variable name. Otherwise, the longest static prefix is the longest prefix of the name that is a static signal or variable name. (8.1) *See also:* **static signal name**.

loop parameter: A constant, implicitly declared by the **for** clause of a loop statement, used to count the number of iterations of a loop. (10.10)

lower bound: The left bound of an ascending range or the right bound of a descending range. (5.2.1)

match: A property of a signature with respect to the parameter and subtype profile of a subprogram or enumeration literal. The signature is said to *match* the parameter and result type profile if certain conditions are true. (4.5.3)

matching case statement: A case statement that includes the question mark delimiter, in which choices are compared with the expression using the “?=” operator. (10.9)

matching elements: Corresponding elements of two composite type values that are used for certain logical and relational operations. (9.2.3)

matching index value: In an element association with a choice that is a discrete range and an expression of the type of the aggregate, the index value in the range that corresponds to a given element of the expression value. (9.3.3.3)

mature callback: A one-time callback whose trigger event has occurred. (21.1)

member: A slice of an object, a subelement, or an object; or a slice of a subelement of an object. (5.1)

metalogical value: One of the enumeration literals 'U', 'X', 'W', or '–' of the type STD_ULOGIC defined in the package STD_LOGIC_1164. (16.8.2.2)

method: An abstract operation that operates atomically and exclusively on a single object of a protected type. (5.6.2)

mode: The direction of information flow through the port or parameter. Modes are **in**, **out**, **inout**, **buffer**, or **linkage**. (6.5.2, 6.5.6.3)

model: The result of the elaboration of a design hierarchy. The *model* can be executed in order to simulate the design it represents. (14.1, 14.7)

model name: An identifier that, jointly with an object library name, uniquely identifies a foreign model. (20.2.2)

modified relative search string: A relative search string modified by the insertion of signatures. (23.22)

most specialized class: That class of which a given object is a member and for which there is no subclass of which the object is also a member. (17.2.1)

multiplicity: The number of permissible target objects of a navigable association. Multiplicities may be 0..1 or 1 for a one-to-one association or 0..* or 1..* for one-to-many associations. (19.2.1)

name: A property of an identifier with respect to some named entity. Each form of declaration associates an identifier with a named entity. In certain places within the scope of a declaration, it is valid to use the identifier to refer to the associated named entity; these places are defined by the visibility rules. At such places, the identifier is said to be the *name* of the named entity. (*6.1*, 8.1)

named association: An association element in which the formal designator appears explicitly. (6.5.7.1, 9.3.3.1)

named entity: An item associated with an identifier, character literal, or operator symbol as the result of an explicit or implicit declaration. (6.1) *See also:* **name**.

navigable: An association in the information model is navigable from a reference object to a target object if it is permissible to acquire a handle for the target object using the `vhpi_handle` function (for a one-to-one association) or the `vhpi_iterator` function (for a one-to-many association) with a handle to the reference object. (19.2.1)

net: A collection of drivers, signals (including ports and implicit signals), conversion functions, and resolution functions that connect different processes. Initialization of a net occurs after elaboration, and a net is updated during each simulation cycle. (14.1, 14.2, *14.7.3.4*)

nonobject alias: An alias whose designator denotes some named entity other than an object. (*6.6.1*, 6.6.3) *See also:* **object alias**.

nonpostponed process: An explicit or implicit process whose source statement does not contain the reserved word **postponed**. When a nonpostponed process is resumed, it executes in the current simulation cycle. Thus, nonpostponed processes have access to the current values of signals, whether or not those values are stable at the current model time. (11.3)

null array: Any of the discrete ranges in the index constraint of an array that define a null range. (5.3.2.2)

null range: A range that specifies an empty subset of values. A range L **to** R is a null range if L > R, and range L **downto** R is a null range if L < R. (5.2.1)

null slice: A slice whose discrete range is a null range. (8.5)

null transaction: A transaction produced by evaluating a null waveform element. (10.5.2.2)

null waveform element: A waveform element that is used to turn off a driver of a guarded signal. (10.5.2.2)

numeric literal: An abstract literal or a literal of a physical type. (9.3.2)

numeric type: An integer type, a floating-point type, or a physical type. (5.2.1)

object: (**A**) A named entity that has a value of a given type. An object can be a constant, signal, variable, or file. (6.4.1) (**B**) An instance of a class in an information model. An object is also an instance of each superclass of the class. (17.2.1)

object alias: An alias whose alias designator denotes an object (that is, a constant, signal, variable, or file). (*6.6.1, 6.6.2*) *See also:* **nonobject alias**.

object callback: A callback whose trigger event relates to the value of a variable or a signal, represented by a trigger object. (21.3.2)

object library: An implementation-defined library containing one or more entry points for elaboration, execution or registration functions. (20.2.2)

one-time callback: A callback for which the callback function is triggered at most once. (21.1)

one-to-many association: An association in which one reference object is associated with possibly more than one target object. (17.2.1)

one-to-one association: An association in which one reference object is associated with at most one target object. (17.2.1)

operation: A function that pertains to a given object or class in an information model. (17.2.1)

optimized contributor: A contributor resulting from an implementation-defined optimization of the representation of contributors and loads of a net. (19.12.2)

optimized load: A load resulting from an implementation-defined optimization of the representation of contributors and loads of a net. (19.12.2)

ordered: A constraint upon a one-to-many association that indicates that an ordering relation applies to the target objects of the association. (19.2.1)

ordinary case statement: A case statement that does not include the question mark delimiter, in which choices are compared with the expression using the "=" operator. (10.9)

out: One possible mode of a port or subprogram parameter. A port of mode **out** contributes a driving value to the network containing the port; the design entity containing the port may also read the port. A subprogram parameter of mode **out** can be modified, and, if it is a variable, its value can be read by the containing subprogram. The value read is the current value of the formal parameter. (4.2.2, 6.5.2, 6.5.6.3)

overloaded: Identifiers or enumeration literals that denote two different named entities. Enumeration literals, subprograms, and predefined operators may be overloaded. At any place where an overloaded enumeration literal occurs in the text of a program, the type of the enumeration literal shall be determinable from the context. (4.2.1, 4.5.1, 4.5.2, 4.5.3, *5.2.2.1*)

parameter: A constant, signal, variable, or file declared in the interface list of a subprogram specification. The characteristics of the class of objects to which a given parameter belongs are also characteristics of the parameter. In addition, a parameter has an associated mode that specifies the direction of dataflow allowed through the parameter. (4.2.2.1, 4.2.2.2, 4.2.2.3, 4.2.2.4, 4.5, 4.8)

parameter and result type profile: Two subprograms that have the same parameter type profile, and either both are functions with the same result base type, or neither of the two is a function. (4.5.1)

parameter interface list: An interface list that declares the parameters for a subprogram. It may contain interface constant declarations, interface signal declarations, interface variable declarations, interface file declarations, or any combination thereof. (6.5.6.1)

parameter type profile: Two formal parameter lists that have the same number of parameters, and at each parameter position the corresponding parameters have the same base type. (4.5.1)

parent: A process or a subprogram that contains a procedure call statement for a given procedure or for a parent of the given procedure. (4.3)

passive process: A process statement where neither the process itself, nor any procedure of which the process is a parent, contains a signal assignment statement. (11.3)

permanent: A permanent string or structure is allocated by the tool in storage that is not subsequently overwritten during the invocation of the tool. A VHPI program may store a pointer to a permanent string or structure for subsequent reference to the string or structure. (23.1)

physical literal: A numeric literal of a physical type. (5.2.4.1)

physical structure: A C struct of type `vhpiPhysT` that represents a value of a physical type. (22.2.6)

physical type: A numeric scalar type that is used to represent measurements of some quantity. Each value of a physical type has a position number that is an integer value. Any value of a physical type is an integral multiple of the primary unit of measurement for that type. (*5.2.1*, 5.2.4)

port: A channel for dynamic communication between a block and its environment. A signal declared in the interface list of an entity declaration, in the header of a block statement, or in the interface list of a component declaration. In addition to the characteristics of signals, ports also have an associated mode; the mode constrains the directions of dataflow allowed through the port. (6.4.2.3, *6.5.6.3*)

port interface list: An interface list that declares the inputs and outputs of a block, component, or design entity. It consists entirely of interface signal declarations. (6.5.6.1, *6.5.6.3*, 11.2)

positional association: An association element that does not contain an explicit appearance of the formal designator. An actual designator at a given position in an association list corresponds to the interface element at the same position in the interface list. (6.5.7.1, 9.3.3.1)

postponed process: An explicit or implicit process whose source statement contains the reserved word **postponed.** When a postponed process is resumed, it does not execute until the final simulation cycle at the current modeled time. Thus, a postponed process accesses the values of signals that are the "stable" values at the current simulated time. (11.3)

predefined operations: Implicitly defined subprograms and predefined operators that operate on the predefined types. (5.2.6, 5.3.2.4, 5.4.3, 5.5.2, 9.2)

predefined operators: Implicitly defined operators that operate on the predefined types. Every predefined operator is a pure function. No predefined operators have named formal parameters; therefore, named association cannot be used in a function whose name denotes a predefined operator. (9.2, 16.3)

primary: One of the elements making up an expression. Each primary has a value and a type. (9.1)

private key: One key of a key pair used with an asymmetric cipher for the encryption or decryption of information. The private key is known only to the owner of the key pair. (24.1.1)

projected output waveform: A sequence of one or more transactions representing the current and projected future values of the driver. (14.7.2)

property: An item of data that pertains to a given object or class in an information model. (17.2.1)

protected type: A type whose objects are protected from simultaneous access by more than one process. (5.6)

protection envelope: A collection of protect tool directives that specify ciphers and keys used to encrypt or decrypt an enclosed portion of a VHDL description. A protection envelope is either an encryption envelope or a decryption envelope. (24.1.1)

public key: One key of a key pair used with an asymmetric cipher for the encryption or decryption of information. The public key is published by the owner of the key pair. (24.1.1)

pulse rejection limit: The threshold time limit for which a signal value whose duration is greater than the limit will be propagated. A pulse rejection limit is specified by the reserved word **reject** in an inertially delayed signal assignment statement. (10.5.2.1)

pure function: A function that returns the same value each time it is called with the same values as actual parameters. An *impure* function may return a different value each time it is called, even when different calls have the same actual parameter values. (4.2.1)

quiet: In a given simulation cycle, a signal that is not active. (14.7.3.1)

range: A specified subset of values of a scalar type. (5.2.1) *See also:* **ascending range**; **belong** (**to a range**)**; descending range**; **left bound**; **lower bound**; **right bound**; **upper bound**.

range constraint: A construct that specifies the range of values in a type. A range constraint is *compatible* with a subtype if each bound of the range belongs to the subtype or if the range constraint defines a null range. The direction of a range constraint is the same as the direction of its range. (*5.2.1*, 5.2.3.1, 5.2.4.1, 5.2.5.1)

read: The value of an object is said to be *read* when its value is referenced or when certain of its attributes are referenced. (6.5.2)

real literal: An abstract literal of the type *universal_real* that contains a base point. (15.5.1)

record type: A composite type whose values consist of named elements. (*5.3.3*, 9.3.3.2)

reference: Access to a named entity. Every appearance of a designator (a name, character literal, or operator symbol) is a reference to the named entity denoted by the designator, unless the designator appears in a library clause or use clause. (12.4, 13.2)

reference class: The class of an object from which a navigable association may be navigated using the `vhpi_handle` or `vhpi_iterator` function. (19.2.1)

reference object: An object from which a navigable association may be navigated using the `vhpi_handle` or `vhpi_iterator` function. (19.2.1)

register: A kind of guarded signal that retains its last driven value when all of its drivers are turned off. (6.4.2.3)

registration: The means whereby a VHPI program identifies a foreign model, foreign application, or callback to the tool so that the tool can invoke the foreign model, application, or callback. (20.2.1, 21.2.2)

registration function: A function in a library of foreign models that performs registration of the foreign models in the library. (21.2.2)

registration phase: That phase of tool execution in which the tool has begun executing, and foreign models and applications are identified to the tool. (20.2)

regular structure: Instances of one or more components arranged and interconnected (via signals) in a repetitive way. Each instance may have characteristics that depend upon its position within the group of instances. Regular structures may be represented through the use of the generate statement. (11.8)

relative search string: A search string provided to the `vhpi_handle_by_name` function that represents a name to be concatenated to the full name of a reference object. (23.2)

release a forced object: An update of a driver, signal, or variable that causes the object no longer to be forced. (22.5)

release a handle: A VHPI program that releases a handle referring to an object indicates to the tool that the VHPI program no longer needs the reference to the object. The tool may reclaim resources used to implement the reference. (17.4.3)

repetitive callback: A callback for which the callback function may be triggered multiple times. (21.1)

reset phase: That phase of tool execution in which a VHDL model is restarted from the state it was in at the end of initialization. (20.9)

resolution: The process of determining the resolved value of a resolved signal based on the values of multiple sources for that signal. (4.6, 6.4.2.3)

resolution function: A user-defined function that computes the resolved value of a resolved signal. (4.6, 6.4.2.3)

resolution limit: The primary unit of type TIME (by default, 1 fs). Any TIME value whose absolute value is smaller than this limit is truncated to zero (0) time units. (5.2.4.2)

resolved signal: A signal that has an associated resolution function. (6.4.2.3)

resolved value: The output of the resolution function associated with the resolved signal, which is determined as a function of the collection of inputs from the multiple sources of the signal. (*4.6*, 6.4.2.3)

resource library: A library containing library units that are referenced within the design unit being analyzed. (13.2)

restart phase: That phase of tool execution in which a previously saved VHDL model is restarted from the point of its save. (20.8)

result subtype: The subtype of the returned value of a function. (4.2.1)

resume: The action of a wait statement upon an enclosing process when the conditions on which the wait statement is waiting are satisfied. If the enclosing process is a nonpostponed process, the process will subsequently execute during the current simulation cycle. Otherwise, the process is a postponed process, which will execute during the final simulation cycle at the current simulated time. (14.7.5)

right bound: For a range L **to** R or L **downto** R, the value R. (5.2.1)

right of: When a value V1 and a value V2 belong to a range and either the range is an ascending range and V2 is the predecessor of V1, or the range is a descending range and V2 is the successor of V1. (5.2.1)

role name: An annotation of a navigable association that identifies that association. (19.2.1)

satisfy: A property of a value with respect to some constraint. The value is said to *satisfy* a constraint if the value is in the subset of values determined by the constraint. (*5.1, 5.3.2.2*)

save phase: That phase of tool execution in which the current state of a VHDL model is saved for possible restart. (20.7)

scalar type: A type whose values have no elements. Scalar types consist of *enumeration types*, *integer types*, *physical types*, and *floating-point types*. Enumeration types and integer types are called *discrete types*. Integer types, floating-point types, and physical types are called *numeric types*. All scalar types are ordered; that is, all relational operators are predefined for their values. (5.1, *5.2*)

schedule a transaction: An update of a driver or a collection of drivers using the `vhpi_schedule_transaction` function to add transactions to the projected output waveforms. (22.6)

scheduled deposit: An update for a driver or signal performed using the `vhpi_put_value` function with an update mode of `vhpiDepositPropagage`. The deposit occurs on the driver or signal on the next signal update phase of a simulation cycle. (14.7.2, 14.7.3, 22.5.3, 22.5.4)

scheduled force: An update for a driver or signal performed using the `vhpi_put_value` function with an update mode of `vhpiForcePropagage`. The driver or signal becomes forced on the next signal update phase of a simulation cycle. (14.7.2, 14.7.3, 22.5.3, 22.5.4)

scope: A portion of the text in which a declaration may be visible. This portion is defined by visibility and overloading rules. (12.2)

secret key: A key used with a symmetric cipher for the encryption and decryption of information. (24.1.1)

selected name: Syntactically, a name having a prefix and suffix separated by a dot. Certain selected names are used to denote record elements or objects denoted by an access value. The remaining selected names are referred to as *expanded names*. (*8.3*, 10.2) *See also:* **expanded name**.

sensitivity set: The set of signals to which a wait statement is sensitive. The sensitivity set is given explicitly in an **on** clause or is implied by an **until** clause. (10.2)

sensitivity-set bitmap: A value structure indicating on which signals in the sensitivity set of a callback an event occurred. (21.3.3.3)

sequential statements: Statements that execute in sequence in the order in which they appear. Sequential statements are used for algorithmic descriptions. (Clause 10)

session key: A key for a symmetric cipher, chosen by an encryption tool for encryption of information in a digital envelope. The session key is encrypted and provided with the encrypted information. (24.1.1)

shared variable: A variable accessible by more than one process. Such variables shall be of a protected type. (6.4.2.4)

short-circuit operation: An operation for which the right operand is evaluated only if the left operand has a certain value. The short-circuit operations are the predefined logical operations **and**, **or**, **nand**, and **nor** for operands of types BIT and BOOLEAN. (9.2.1)

signal: An object with a past history of values. A signal may have multiple drivers, each with a current value and projected future values. The term *signal* refers to objects declared by signal declarations or port declarations. (6.4.2.3)

signal transform: A sequential statement within a statement transform that determines which one of the alternative waveforms, if any, is to be assigned to an output signal. A signal transform is the same simple, conditional, or selected signal assignment statement as is contained in the concurrent signal assignment statement for which the statement transform is defined. (11.6)

simple name: The identifier associated with a named entity, either in its own declaration or in an alias declaration. (8.2)

simple package: A package declared by a package declaration containing no generic clause and no generic map aspect. (4.7)

simple subprogram: A subprogram declared by a subprogram declaration containing no generic list and no generic map aspect. (4.2.1)

simulated net: A set of of objects, represented by objects of class `signal`, that have the same effective and driving values, as appropriate, at all simulations times. (19.12.2)

simulation cycle: One iteration in the repetitive execution of the processes defined by process statements in a model. The first simulation cycle occurs after initialization. A simulation cycle can be a delta cycle or a time-advance cycle. (14.7.5)

simulation phase: That phase of tool execution in which execution of an elaborated and initialized design hierarchy occurs. (14.7.5.3, *20.6*)

simulation phase callback: A callback whose trigger event relates to steps of the simulation cycle. (21.3.6)

single-line comment: A comment that starts with two adjacent hyphens and extends up to the end of the line. (15.9)

single-object declaration: An object declaration whose identifier list contains a single identifier; it is called a *multiple-object declaration* if the identifier list contains two or more identifiers. (6.4.2.1)

slice: A one-dimensional array of a sequence of consecutive elements of another one-dimensional array. (8.5)

source: A contributor to the value of a signal. A source can be a driver or port of a block with which a signal is associated or a composite collection of sources. (6.4.2.3)

specification: A class of construct that associates additional information with a named entity. There are three kinds of specifications: attribute specifications, configuration specifications, and disconnection specifications. (Clause 7)

standard direct binding: A form of foreign attribute value that specifies an object library path, elaboration function name, and execution function name for a foreign model. (20.2.4.3)

standard indirect binding: A form of foreign attribute value that specifies an object library name and a model name. The tool uses the foreign attribute value in conjunction with registration information to locate the elaboration and executions functions for the foreign model. (20.2.4.2)

statement callback: A callback whose trigger event relates to execution of one or more statements of suspension or resumption of a process. (21.3.4)

statement transform: The first sequential statement in the process equivalent to the concurrent signal assignment statement. The statement transform defines the actions of the concurrent signal assignment statement when it executes. The statement transform is followed by a wait statement, which is the final statement in the equivalent process. (11.6)

static: *See:* **globally static expression**; **globally static primary**; **locally static expression**; **locally static name**; **locally static primary**; **locally static subtype**.

static name: A name in which every expression that appears as part of the name (for example, as an index expression) is a static expression (if every discrete range that appears as part of the name denotes a static range or subtype and if no prefix within the name is either an object or value of an access type or a function call). (8.1)

static object: An object in an information model that, once created, remains in existence until termination of the tool. (17.2.1)

static range: A range whose bounds are static expressions. (9.4)

static signal name: A static name that denotes a signal. (8.1)

static variable name: A static name that denotes a variable. (8.1)

string literal: A sequence of graphic characters, or possibly none, enclosed between two quotation marks ("). The type of a string literal is determined from the context. (9.3.2, 15.7)

string representation: A string that represents the value of a given type. A string representation of a value is returned by the TO_STRING operation. (5.7)

subaggregate: An aggregate appearing as the expression in an element association within another, multidimensional array aggregate. The subaggregate is an (n–1)-dimensional array aggregate, where n is the dimensionality of the outer aggregate. Aggregates of multidimensional arrays are expressed in row-major (right-most index varies fastest) order. (9.3.3.3)

subclass: The class in an inheritance relationship that inherits properties, operations, and associations. (17.2.1)

subelement: An element of another element. Where other subelements are excluded, the term *element* is used. (5.1)

subprogram specification: Specifies the designator of the subprogram, any formal parameters of the subprogram, and the result type for a function subprogram. (4.2.1)

subtype: A type together with a constraint. A value *belongs* to a subtype of a given type if it belongs to the type and satisfies the constraint; the given type is called the *base type* of the subtype. A type is a subtype of itself. Such a subtype is said to be *unconstrained* because it corresponds to a condition that imposes no restriction. A subtype S1 is *compatible* with a subtype S2 if the range constraint associated with S1 is compatible with S2. (5.1)

superclass: The class in an inheritance relationship from which properties, operations, and associations are inherited. (17.2.1)

suspend: A process that stops executing and waits for an event or for a time period to elapse. (14.7.5)

symmetric cipher: A cipher requiring a single key, called the *secret key,* for both encryption and decryption of information. (24.1.1, 24.1.3.2)

synthesis tool: Any tool that interprets VHDL source code as a description of an electronic circuit in accordance with the terms of this standard and derives an alternate description of that circuit. (16.8.1.2)

tabular registry: A text file containing the registration information for foreign models and applications. (20.2.2)

target class: The class of an object to which navigation via a navigable association is permitted using the `vhpi_handle` or `vhpi_iterator` function. (19.2.1)

target library: A library containing the design unit in which a given component is declared. The target library is used to determine the visible entity declaration under certain circumstances for a default binding indication (7.3.3)

target object: An object to which navigation via a navigable association is permitted using the `vhpi_handle` or `vhpi_iterator` function. (19.2.1)

termination phase: That phase of tool execution in which the tool has completed execution and is terminating. (20.10)

time callback: A callback whose trigger event relates to progress of simulation time. (21.3.5)

time structure: A C struct of type `vhpiTimeT` that represents a non-negative time. (22.2.7)

timeout interval: The maximum time a process will be suspended, as specified by the timeout period in the until clause of a wait statement. (10.2)

to the left of: *See:* **left of**.

to the right of: *See:* **right of**.

tool: A program that maintains a representation of a VHDL model and provides the VHPI functions. (17.2.1)

transaction: A pair consisting of a value and a time. The value represents a (current or) future value of the driver; the time represents the relative delay before the value becomes the current value. (14.7.2)

transient: A transient string or structure is allocated by the tool in storage that may subsequently be overwritten. The value of the string or structure persists at least until the earlier of the next call to the given VHPI function by the same thread of control or the return to the tool by the thread of control that called the given VHPI function. If a VHPI program needs to refer to the value of a transient string or structure beyond the interval for which it persists, the VHPI program shall copy the value. (23.1)

transport delay: An optional delay model for signal assignment. Transport delay is characteristic of hardware devices (such as transmission lines) that exhibit nearly infinite frequency response: any pulse is transmitted, no matter how short its duration. (10.5.2.1) *See also:* **inertial delay**.

trigger event: An occurrence of a callback reason that causes a callback, if enabled, to be invoked. (21.1)

trigger object: An object in an information model that is associated with a trigger event for a callback. (21.1)

type: A set of values and a set of operations. (Clause 5)

type conversion: An expression that converts the value of a subexpression from one type to the designated type of the type conversion. Associations in the form of a type conversion are also allowed. These associations have functions and restrictions similar to conversion functions but can be used in places where conversion functions cannot. In both cases (expressions and associations), the converted type shall be closely related to the designated type. (6.5.7.1, 9.3.6) *See also:* **closely related types; conversion function**.

unaffected: A waveform in a signal assignment statement that does not affect the driver of the target. (*10.5.2.1*)

unassociated formal: A formal that is not associated with an actual. (6.5.7.2, 6.5.7.3)

unconstrained subtype: A subtype that corresponds to a condition that imposes no restriction. (*5.1*, 6.3)

uninstantiated package: A package declared by a package declaration containing a generic clause and no generic map aspect. An uninstantiated package may be instantiated with a package instantiation declaration. (4.7, 4.9)

uninstantiated subprogram: A subprogram declared by a subprogram declaration containing a generic list and no generic map aspect. An uninstantiated subprogram may be instantiated with a subprogram instantiation declaration. (4.2.1, 4.4)

unit name: A name defined by a unit declaration (either the primary unit declaration or a secondary unit declaration) in a physical type declaration. (5.2.4.1)

universal_integer: An anonymous predefined integer type that is used for all integer literals. The position number of an integer value is the corresponding value of the type *universal_integer*. (*5.2.3.1*, 9.3.2, 9.3.6)

universal_real: An anonymous predefined type that is used for literals of floating-point types. Other floating-point types have no literals. However, for each floating-point type there exists an implicit conversion that converts a value of type *universal_real* into the corresponding value (if any) of the floating-point type. (*5.2.3.1*, 9.3.2, 9.3.6)

update: An action on the value of a signal, variable, or file. The value of a signal is said to be *updated* when the signal appears as the target (or a element of the target) of a signal assignment statement (indirectly); when it is associated with an interface object of mode **out**, **buffer**, **inout**, or **linkage**; or when one of its subelements (individually or as part of a slice) is updated. The value of a signal is also said to be *updated* when it is a subelement or slice of a resolved signal, and the resolved signal is updated. The value of a variable is said to be *updated* when the variable appears as the target (or a element of the target) of a variable assignment statement (indirectly), when it is associated with an interface object of mode **out** or **linkage**, or when one of its subelements (individually or as part of a slice) is updated. The value of a file is said to be *updated* when a WRITE or FLUSH operation is performed on the file object. (6.5.2)

upper bound: The right bound of an ascending range or the left bound of a descending range. (5.2.1)

valid handle: A handle that refers to an object that exists. (17.4.5)

value structure: A C struct of type `vhpiValueT` that represents a scalar value, a one-dimensional array of scalar values, or a value of any type represented in an implementation-defined internal representation. (22.2.8)

variable: An object with a single current value. (6.4.2.4)

VHPI program: A program that calls the VHPI functions. (17.2.1)

visible: When the declaration of an identifier defines a possible meaning of an occurrence of the identifier used in the declaration. A visible declaration is visible by selection (for example, by using an expanded name) or directly visible (for example, by using a simple name). (12.3)

visible entity declaration: The entity declaration selected for default binding in the absence of explicit binding information for a given component instance. (7.3.3)

waveform: A series of transactions, each of which represents a future value of the driver of a signal. The transactions in a waveform are ordered with respect to time, so that one transaction appears before another if the first represents a value that will occur sooner than the value represented by the other. (10.5.2.1)

whitespace character: A space, a nonbreaking space, or a horizontal tabulation character (SP, NBSP, or HT). (16.4)

working library: A design library into which the library unit resulting from the analysis of a design unit is placed. (13.2)

Annex J

(informative)

Bibliography

[B1] Anderson, Ross, Biham, Eli, and Knudsen, Lars, *Serpent: A Proposal for the Advanced Encryption Standard;* available at http://www.cl.cam.ac.uk/ftp/users/rja14/serpent.tar.gz.

[B2] ANSI X9.52-1998, American National Standard for Financial Services—Triple Data Encryption Algorithm Modes of Operation.[19]

[B3] ElGamal, Taher, "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *IEEE Transactions on Information Theory,* vol. IT-31, no. 4, 1985, pp. 469–472.

[B4] FIPS PUB 46-3, Data Encryption Standard (DES), 1999 October 25.[20]

[B5] FIPS PUB 81, DES Modes of Operation, 1980 December 25.

[B6] FIPS PUB 180-3, Secure Hash Standard (SHS), October 2008.

[B7] FIPS PUB 197, Advanced Encryption Standard (AES), 2001 November 26.

[B8] IEEE 100, *The Authoritative Dictionary of IEEE Standards Terms,* Seventh Edition. New York: Institute of Electrical and Electronics Engineers, Inc.[21, 22]

[B9] IEEE Std 1003.1™, 2004 Edition, IEEE Standard for Information Technology—Portable Operating System Interface (POSIX™).

[B10] IEEE Std 1076.1™-2007, IEEE Standard VHDL Analog and Mixed-Signal Extensions.

[B11] IEEE Std 1076.2™-1996, IEEE Standard VHDL Mathematical Packages.

[B12] IEEE Std 1076.3™-1997, IEEE Standard VHDL Synthesis Packages.

[B13] IEEE Std 1076.4™-2000, IEEE Standard for VITAL Application-Specific Integrated Circuit (ASIC) Modeling Specification.

[B14] IEEE Std 1076.6™-2004, IEEE Standard for VHDL Register-Transfer Level (RTL) Synthesis.

[B15] IEEE Std 1149.1™-2001, IEEE Standard Test Access Port and Boundary Scan Architecture.

[B16] IEEE Std 1164™-1993, IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164).

[19]ANSI publications are available from the Sales Department, American National Standards Institute, 25 West 43rd Street, 4th Floor, New York, NY 10036, USA (http://www.ansi.org/).

[20]FIPS publications are available from the National Technical Information Service (NTIS), U. S. Dept. of Commerce, 5285 Port Royal Rd., Springfield, VA 22161 (http://www.ntis.org/); also available at http://csrc.nist.gov/publications/PubsFIPS.htlm.

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

[22]The IEEE standards or products referred to in this clause are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

[B17] IETF RFC 1319, The MD2 Message-Digest Algorithm, April 1992.[23]

[B18] IETF RFC 1321, The MD5 Message-Digest Algorithm, April 1992.

[B19] IETF RFC 2045, Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies, November 1996.

[B20] IETF RFC 2144, The CAST-128 Encryption Algorithm, May 1997.

[B21] IETF RFC 2437, PKCS #1: RSA Cryptography Specifications, Version 2.0, October 1998.

[B22] IETF RFC 2440, OpenPGP Message Format, November 1998.

[B23] ISO/IEC 10118-3:2004, Information technology—Security techniques—Hash-functions—Part 3: Dedicated hash-functions.[24]

[B24] Schneier, Bruce, "Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)," *Fast Software Encryption, Cambridge Security Workshop Proceedings* (December 1993), Springer-Verlag, 1994, pp. 191–204.

[B25] Schneier, Bruce et al., *The Twofish Encryption Algorithm.* Wiley, 1999.

[23]IETF publications are available from http://www.ietf.org/rfc.html.

[24]ISO/IEC publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse (http://www.iso.ch/). ISO/IEC publications are also available in the United States from Global Engineering Documents, 15 Inverness Way East, Englewood, Colorado 80112, USA (http://global.ihs.com/). Electronic copies are available in the United States from the American National Standards Institute, 25 West 43rd Street, 4th Floor, New York, NY 10036, USA (http://www.ansi.org/).

Index

Page numbers in **boldface** indicate references to clauses and subclauses. Page numbers in *italics* indicate occurrence of the term in a BNF syntax rule. In most cases, the term is further described in text following the syntax rule.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z