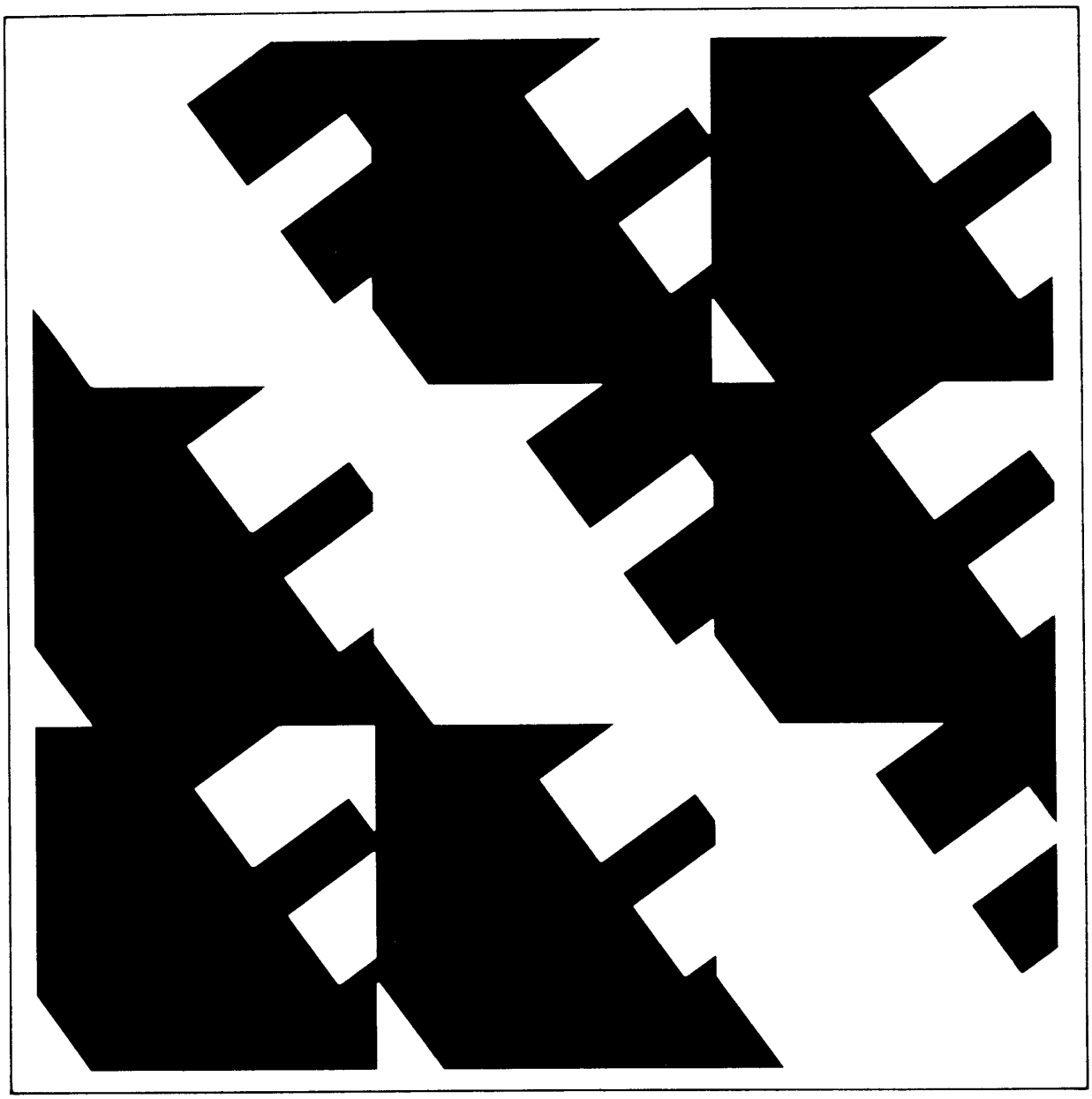
IEEE Standard VHDL Language Reference Manual



IEEE Std 1076-1987

Published by The Institute of Electrical and Electronics Engineers, Inc 345 East 47th Street, New York, NY 10017, USA

March 31, 1988

SH11957

IEEE
Std 1076-1987

IEEE Standard VHDL

Language Reference Manual

Sponsors

Design Automation Standards Subcommittee
of the
Design Automation Technical Committee
of the
Computer Society of the IEEE

and

Automatic Test Program Generation Subcommittee
of the
IEEE Standards Coordinating Committee 20

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

Foreword

(This Foreword is not a part of IEEE Std 1076-1987, IEEE Standard VHDL Language Reference Manual.)

The VHDL standardization effort has been supported by volunteers with expertise in computer systems design and manufacturing, aerospace, communications, CAD tool development, IC model development and other areas. The names of the participants in the VASG meetings are given in the preface. Many of the VASG volunteers have made major contributions to the analysis of VHDL requirements, the analysis of language issues, and the review of the design of the language.

The final language design presented in this document is the result of the efforts of Erich Marschner and Moe Shahdad, the principal designers of VHDL. Their hard work and professionalism contributed significantly to the final result. Their dedication is to be applauded.

Ronald Waxman
Chairperson, DASS

Larry Saunders
Chairperson, VASG

Preface

(This Preface is not a part of IEEE Std 1076-1987, 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.

Development of IEEE Standard VHDL

IEEE Standard VHDL was developed through the work of the VHDL Analysis and Standardization Group (VASG), a working group within the Design Automation Standards Subcommittee (DASS) of the Design Automation Technical Committee (DATC) of the Computer Society of the IEEE. The work of the VASG was jointly sponsored by the DATC and by the Automatic Test Program Generation (ATPG) subcommittee of the IEEE Standards Coordinating Committee 20 (SCC20). Mr. Larry Saunders is the Chairman of the VASG; Mr. Ron Waxman is Chairman of the DASS; Mr. Al Lowenstein is the Chairman of the ATPG subcommittee.

The standardization of VHDL began in February 1986 with the adoption of VHDL version 7.2, described below, as the baseline language. In order to assist the voluntary standardization process of the IEEE, the Air Force Wright Aeronautical Laboratories contracted with CAD Language Systems Inc. (CLSI) to support the IEEE in the analysis of VHDL language issues, extension of the baseline language, and preparation of the draft and final definitions of the IEEE Standard VHDL. This work was performed under contracts F33615-82-C-1716 and F33615-86-C-1050.

CLSI prepared a series of analysis and recommendation reports, which were presented to the VASG at eight industry meetings over a period of nine months. A draft standard was issued at the end of 1986, and following a review, a second draft was issued and balloted in mid-1987. The resulting standard reflects the adoption of recommendations made by CLSI, by members of the VASG, and by others who contributed to the standardization effort.

The CLSI Project Manager for the IEEE standardization effort was Dr. Moe Shahdad, and the CLSI Technical Lead for the development of IEEE Standard VHDL was Mr. Erich Marschner. The Air Force point of contact for the IEEE VHDL standardization effort was Dr. John Hines. Ron Waxman, Chairman of the DASS, was the IEEE coordinator.

Many individuals from many different organizations participated in the development of IEEE Standard VHDL. In particular, the following people attended meetings of the VASG:

Dean Anderson
Kevin Anderson
Larry Anderson
Jim Armstrong
Lisa Asher
James Aylor
Jwahar Bammi
Peter Barck
Daniel Barclay
Dave Barton
Bill Beck
Victor Berman
Ken Caron
Hal Carter
Marc Casad
Moon Jung Chung
Patti Cochran
Dave Coelho
Doug Dunlop
Cathy Edwards
Thomas Elliot
Mike Endrizzi
Dave Evans
Deborah Frauenfelder
Mark Glewwe
Prabhu Goel
William Guzek
Jeff Haeffele
Charlie Haynes
John Hines
Mike Hirasuna
Ray Hookway
Ching Hsiao
Paul Hubbard
Youm Huh
John Jensen

Bob Johnson
Susan Johnston
George Konstantinow
Stan Krolikoski
Rick Lazansky
Jean Lester
Roger Lipsett
Shin-ming Liu
Al Lowenstein
Bruce Lundeby
Mark Macke
Robert Mackey
Erich Marschner
Paul Menchini
Lynn Meredith
Jean Mermet
Ellen Mickanin
Kieu Mien Le
Dwight Miller
Kent Moffat
Bob Morris
Jim Morris
Dan Nash
John Newkirk
Tim Noble
Ghulam Nurie
Leslie Orlidge
Ed Ott
Thomas Panfil
Steve Piatz
Signe Post
Jean Pouilly
Bob Powell
Kim Rawlinson
Joel Rodriguez
Cary Sandvig
Larry Saunders
Lowell Savage
Tim Saxe
Dick Schlotfeldt
Peggy Schmidt
Ken Scott
Moe Shahdad
Arina Shainski
Alec Stanculescu
Stephen Sutherland
Tom Tempero
Jacques Tete
Tim Thorp
Tuan Tran
Stan Wagner
Rich Wallace
Karen Watkins
Ron Waxman
Isaiah White
Greg Winter
Craig Winton
Dan Youngbauer

IEEE Standard 1076-1987 is being maintained by the VHDL Analysis and Standardization Group, a working group of the Design Automation Standards Subcommittee. This group has been established to resolve issues that may arise with the language and to consider potential extensions to the language. The working documents generated by the VASG will be available from the Computer Society Standards Secretariat, Computer Society of the IEEE, 1730 Massachusetts Ave. N.W., Washington, DC 20036, 202-371-0101, and also from the Computer Society Secretariat, IEEE Standards Office, 345 East 47th Street, New York, NY 10017, 212-705-7960. The working documents are not formally approved documents; however, they do reflect current status of the working group's direction. If you have an interest in the issues related to this standard, please contact the Chair of the VASG.

Documents Relating to IEEE Standard VHDL

In addition to this standard, a number of other documents were developed during the standardization activity. These include:

- The Proceedings of the VHDL Analysis and Standardization Group. These Proceedings include the VASG Analysis and Recommendation reports prepared by CLSI and the minutes of the VASG meetings.

- The IEEE Standard VHDL Language Refinement Rationale. The Language Refinement Rationale explains the reasons for the adopted changes, both as compared to the baseline language (VHDL 7.2) and as compared to other changes that were proposed but not adopted.

- The IEEE Standard VHDL Tutorial. The Tutorial makes extensive use of real hardware design examples to present the language in terms of its relevance to hardware design problems.

The Tutorial also illustrates how information expressed in the Electronic Design Interchange Format (EDIF) can be represented within VHDL. These documents are available from CLSI.

Conventions Used in This Standard

The form of a VHDL description is described by means of context-free syntax together with context-dependent syntactic and semantic requirements expressed by narrative rules. The context free syntax of the language is described using a simple variant of Backus-Naur Form, in particular:

(a) Lower case words, 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).

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

array

(c) A vertical bar separates alternative items unless it occurs immediately after an opening brace, in which case it stands for itself:

letter_or_digit ::= letter | digit

choices ::= choice { | choice}

(d) Square brackets enclose optional items, thus the two following rules are equivalent:

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

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

(e) Braces enclose a repeated item. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus the following two rules are equivalent:

term ::= factor {multiplying_operator factor}

term ::= factor | term multiplying_operator factor

(f) 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 equivalent to name alone.

(g) An italicized term in the text indicates definition of that term.

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

Some sections of this standard contain examples and notes. Examples are meant to illustrate the possible forms of the construct described. Notes are meant to emphasize consequences of the rules described in the section or elsewhere. Examples and notes are not part of the definition of the language.

Development of VHDL version 7.2

The development of the VHDL version 7.2 language and support environment were sponsored by the US Air Force under contract F33615-83-C-1003, and commenced in August 1983. Intermediate versions of the language definition were reviewed extensively by the Government, as well as by industry and academia, at various points during the development program. The results of these reviews, as well as feedback from within the contractor team resulting from the tool implementations, resulted in VHDL version 7.2.

The team of companies developing VHDL consisted of Intermetrics, International Business Machines, and Texas Instruments, with Intermetrics as the prime contractor. The overall VHDL program manager was Roger Lipsett of Intermetrics. Moe Shahdad of Intermetrics was the chief language designer for the VHDL language definition effort. Erich Marschner of Intermetrics provided many of the creative ideas embodied in the language definition. Other major contributions were made by Howard Cohen, Doug Dunlop, Alfred Gilman, and Kellye Sheehan of Intermetrics; by Dave Ackley and Don Newman of Texas Instruments; by Leon Maissel, Larry Saunders, and Ron Waxman of International Business Machines; and by Hillel Ofek of Silvar-Lisco.

Many other individuals contributed to the result with their comments, suggestions, and criticism, both as part of internal language reviews and during coding of the benchmarks. These people included: P. Belmont, M. Brown, W. Carlson, J. Crowley, F. Deitz, V. Donaldson, M. Eskew, M. Gordon, E. Hassler, W. Johnson, C. Kronke, L. McCalla, K. Michael, H. Mills, A. Savkar, C. Scarratt, C. Schaefer, E. Skuldt, E. Wasser, and R. Winter.

Finally, much assistance and guidance were provided by John Carnegie, the Texas Instruments VHDL Program Manager, and Phil Johnson, the International Business Machines VHDL Program Manager, as well as Capt. Allen M. Dewey, the COTR for the Air Force, and the VHDL Tri-Service Committee.

Documents Relating to VHDL 7.2

The VHDL 7.2 Language Reference Manual is a formal description of VHDL version 7.2. Other documents that accompany the Language Reference Manual and describe the capabilities of the language with different emphasis are:

- The VHDL User's Manual, containing a VHDL tutorial, a VHDL reference guide, examples of benchmarks coded in VHDL, and a set of usage scenarios showing the ways in which the VHDL system may be employed to perform a variety of functions.

- The VHDL Design Analysis and Justification, which discusses key language design decisions and justifies the choices made.

Certain VHDL constructs are either directly taken from Ada[1] or resemble those of Ada. A discussion of similarities and differences between Ada and VHDL is provided as part of the Design Analysis and Justification Document.

---

[1] Ada is a trademark of the US DoD.

The following persons were on the balloting committee that approved this document for submission to the IEEE Standards Board:

Donald H. Abernathie
Guy Adam
Gordon Adshead
Harvey Alperin
Kenneth Anderson
James R. Armstrong
Lisa R. Asher
Morris Balamut
Peter E. Barck
Daniel S. Barclay
David Barton
Joseph C. Batz
Rodolfo Betancourt
William D. Billowitch
Lester I. Birman
Richard J. Bonneau
Douglas B. Boyle
Craig A. Brown
R. Scott Brunton
Antonius Bunsen
Raul Camposano
Lawrence Carpenter
William N. Carr
Harold W. Carter
Michael T. Carter
Ralph Cavin III
Edmund Cheng
Charles R. Childress
Shiu-Kai Chin
James B. Clary
David Coelho
Frank Conforti
Tedd Corman
James E. Cottrell
Scott H. Cravens
Mr Jack L. Cross
Brian Dalio
Michael W. Davis
Allen M. Dewey
Bernd Dinklage
James Do
Gary Duggan
Leo Egan, Jr.
Ludwig D. Eggermont
Bernd F. Eichenauer
Abdullah C. Erdal
Wolf-Dieter Erdmann
Jim Flournoy
Don Fraser
Hingsum S. Fung
Daniel Gale
Daniel Garvin
Sumit Ghosh
Gerald Ginsberg
Michael J. Gooding
Donald F. Gorman
John Graham
Arnold Greenspan
T. Winston Griffin
J. Steven Grout
Hans Grundner
Jeffrey Haeffele
Arthur Hann
Fred H. Harder
Michael G. Harrison
Raymond L. Hasenstab
Raymond L. Heather
Robert Heatherington
Kenneth E. Hermance
Jack Hilibrand
John Hines
Michael Hirasuna
Raymond Hookway
Paul W. Horstmann
Ching Hsiao
Paul E. Hubbard
Youm Huh
John E. Jensen
Robert Johnson
Timothy V. Johnson
William A. Johnson
Susan Johnston
Joe Kagenski
Osamu Karatsu
Helmuth M. Kaunzinger
Eskil Kjelkerud
Berrie C. Knott
Stanley Krolikoski
Mike Kuzemchak
Sonja V. Kval
Tuvia Lamdan
Glen G. Langdon, Jr.
Robert P. Larsen
Lars-Olov Larsson
Edwin Law
Rick Lazansky
Arthur E. Levy
Jeff Lewis
John Lewis
Charles W. Lillie
Dan Lorts
David B. Loveman
Al Lowenstein
Joseph F. P. Luhukay
Chidchanok Lursinsap
Leon I. Maissel
Fred Malver
Erich Marschner
Arthur J. Mason
Lawrence Maturo
Michael C. McFarland
Robert J. McGowen
Paul J. Menchini
Jean Mermet
Alexander Miczo
Viju Monie
Kris Moorthi
David Morris
Robert Morris
John W. Mowery
Lawrence J. O'Connell
Eckhard Obel
Roy Oishi
Leslie A. Orlidge
Catherine Ozenfant
Rich Palmer
Alice C. Parker
Ken Parker
Curtis Parks
Latit Patnaik
Richard J. Patrick
Frederick D. Petry
Steve Piatz
Jean Pouilly
Paolo Prinetto
Giorgio Puggelli
Edward G. Pumphrey
Richard Rath
Kim Rawlinson
James R. Reeder
Hassan K. Reghbati
Johannes Reh
Daniel Rosenkrantz
Donald E. Rudisill
William E. Russell, Jr
Roy L. Russo
Jere L. Sanburn
Albert Sanson
Debabrata Sarma
Paul R. Saucier
Larry Saunders
Lowell Savage
Anil D. Savkar
Helmut Scheibenzuber
Warren M. Scheinin
Steven Schlosser
Dick Schlotfeld
Moe Shahdad
Ravi Shankar
William H. Sherwood
Isao Shirakawa
Sajjan G. Shiva
Ronald J. Short
David M. Siefert
James E. Sigler
Gabriel M. Silberman
Evangelos Simoudis
Alec Stanculescu
Richard Summar
Tom H. Tabb
Jacques Tete
Tim L. Thorp
Robert Tieche
William C. Topf
Tuan Tran
Jack Trautman
Steve Trimberger
Joseph Tront
Richard W. Tucker
Mark-Rene Uchida
Kenneth Van Bree
Anthony A. F. Vogelpoel
Rene J. Vuarnoz
L. W. Wagner
Ron Waxman
Richard J. Weger
John Michael Williams
Ronald D. Williams
Craig Winton
Chris J. Xydes
Jon J. Yerger
George W. Zobrist

When the IEEE Standards Board approved this standard on December 10, 1987, it had the following membership:

Donald C. Fleckenstein, *Chairman* **Marco W. Migliaro,** *Vice Chairman*
Andrew G. Salem, *Secretary*

James H. Beall
Dennis Bodson
Marshall L. Cain
James M. Daly
Stephen R. Dillon
Eugene P. Fogarty
Jay Forster
Kenneth D. Hendrix
Irvin N. Howell
Leslie R. Kerr
Jack Kinn
Irving Kolodny
Joseph L. Koepfinger*
Edward Lohse
John May
Lawrence V. McCall
L. Bruce McClung
Donald T. Michael*
L. John Rankine
John P. Riganati
Gary S. Robinson
Frank L. Rose
Robert E. Rountree
Sava I. Sherr*
William R. Tackaberry
William B. Wilkens
Helen M. Wood

* Member emeritus

Contents

CHAPTER 1 DESIGN ENTITIES AND CONFIGURATIONS

1.1 Entity Declarations 1-1
1.1.1 Entity Header 1-2
1.1.1.1 Generics 1-3
1.1.1.2 Ports 1-3
1.1.2 Entity Declarative Part 1-4
1.1.3 Entity Statement Part 1-5
1.2 Architecture Bodies 1-6
1.2.1 Architecture Declarative Part 1-7
1.2.2 Architecture Statement Part 1-7
1.3 Configuration Declarations 1-9
1.3.1 Block Configuration 1-10
1.3.2 Component Configuration 1-12

CHAPTER 2 SUBPROGRAMS AND PACKAGES

2.1 Subprogram Declarations 2-1
2.1.1 Formal Parameters 2-2
2.1.1.1 Constant and Variable Parameters 2-2
2.1.1.2 Signal Parameters 2-3
2.2 Subprogram Bodies 2-4
2.3 Subprogram Overloading 2-5
2.3.1 Operator Overloading 2-6
2.4 Resolution Functions 2-7
2.5 Package Declarations 2-8
2.6 Package Bodies 2-10
2.7 Conformance Rules 2-11

CHAPTER 3 TYPES

3.1 Scalar Types 3-2
3.1.1 Enumeration Types 3-3
3.1.1.1 Predefined Enumeration Types 3-3
3.1.2 Integer Types 3-4
3.1.2.1 Predefined Integer Types 3-5
3.1.3 Physical Types 3-5
3.1.3.1 Predefined Physical Types 3-7
3.1.4 Floating Point Types 3-7
3.1.4.1 Predefined Floating Point Types 3-8

3.2	Composite Types	3-8
3.2.1	Array Types	3-8
3.2.1.1	Index Constraints and Discrete Ranges	3-10
3.2.1.2	Predefined Array Types	3-12
3.2.2	Record Types	3-13
3.3	Access Types	3-13
3.3.1	Incomplete Type Declarations	3-14
3.3.2	Allocation and Deallocation of Objects	3-15
3.4	File Types	3-16
3.4.1	File Operations	3-16

CHAPTER 4 DECLARATIONS

4.1	Type Declarations	4-1
4.2	Subtype Declarations	4-2
4.3	Objects	4-3
4.3.1	Object Declarations	4-4
4.3.1.1	Constant Declarations	4-4
4.3.1.2	Signal Declarations	4-5
4.3.1.3	Variable Declarations	4-7
4.3.2	File Declarations	4-7
4.3.3	Interface Declarations	4-8
4.3.3.1	Interface Lists	4-10
4.3.3.2	Association Lists	4-11
4.3.4	Alias Declarations	4-13
4.4	Attribute Declarations	4-14
4.5	Component Declarations	4-15

CHAPTER 5 SPECIFICATIONS

5.1	Attribute Specification	5-1
5.2	Configuration Specification	5-3
5.2.1	Binding Indication	5-4
5.2.1.1	Entity Aspect	5-4
5.2.1.2	Generic Map and Port Map Aspects	5-5
5.2.2	Default Binding Indication	5-6
5.3	Disconnection Specification	5-7

CHAPTER 6 NAMES

6.1	Names	6-1
6.2	Simple Names	6-2
6.3	Selected Names	6-2
6.4	Indexed Names	6-3
6.5	Slice Names	6-4
6.6	Attribute Names	6-4

CHAPTER 7 EXPRESSIONS

7.1 Expressions 7-1
7.2 Operators 7-2
7.2.1 Logical Operators 7-2
7.2.2 Relational Operators 7-3
7.2.3 Adding Operators 7-4
7.2.4 Multiplying Operators 7-6
7.2.5 Miscellaneous Operators 7-7
7.3 Operands 7-8
7.3.1 Literals 7-8
7.3.2 Aggregates 7-9
7.3.2.1 Record Aggregates 7-10
7.3.2.2 Array Aggregates 7-10
7.3.3 Function Calls 7-11
7.3.4 Qualified Expressions 7-12
7.3.5 Type Conversions 7-12
7.3.6 Allocators 7-14
7.4 Static Expressions 7-15
7.5 Universal Expressions 7-17

CHAPTER 8 SEQUENTIAL STATEMENTS

8.1 Wait Statement 8-1
8.2 Assertion Statement 8-2
8.3 Signal Assignment Statement 8-3
8.3.1 Updating a Projected Output Waveform 8-4
8.4 Variable Assignment Statement 8-6
8.4.1 Array Variable Assignments 8-7
8.5 Procedure Call Statement 8-7
8.6 If Statement 8-8
8.7 Case Statement 8-8
8.8 Loop Statement 8-9
8.9 Next Statement 8-10
8.10 Exit Statement 8-10
8.11 Return Statement 8-11
8.12 Null Statement 8-11

CHAPTER 9 CONCURRENT STATEMENTS

9.1 Block Statement 9-1
9.2 Process Statement 9-3
9.2.1 Drivers 9-4
9.3 Concurrent Procedure Call 9-4
9.4 Concurrent Assertion Statement 9-5
9.5 Concurrent Signal Assignment Statements 9-6
9.5.1 Conditional Signal Assignment 9-8
9.5.2 Selected Signal Assignment 9-9
9.6 Component Instantiation Statement 9-10
9.6.1 Instantiation of a Component 9-11
9.7 Generate Statement 9-13

CHAPTER 10 SCOPE AND VISIBILITY

10.1 Declarative Regions 10-1
10.2 Scope of Declarations 10-2
10.3 Visibility 10-3
10.4 Use Clauses 10-5
10.5 The Context of Overload Resolution 10-6

CHAPTER 11 DESIGN UNITS AND THEIR ANALYSIS

11.1 Design Units 11-1
11.2 Design Libraries 11-2
11.3 Context Clauses 11-3
11.4 Order of Analysis 11-3

CHAPTER 12 ELABORATION AND EXECUTION

12.1 Elaboration of a Design Hierarchy 12-1
12.2 Elaboration of a Block Header 12-2
12.2.1 The Generic Clause 12-2
12.2.2 The Generic Map Clause 12-2
12.2.3 The Port Clause 12-2
12.2.4 The Port Map Clause 12-2
12.3 Elaboration of a Declarative Part 12-3
12.3.1 Elaboration of a Declaration 12-3
12.3.1.1 Subprogram Declarations and Bodies 12-3
12.3.1.2 Type Declarations 12-3
12.3.1.3 Subtype Declarations 12-4
12.3.1.4 Object Declarations 12-4
12.3.1.5 Alias Declarations 12-5
12.3.1.6 Attribute Declarations 12-5
12.3.1.7 Component Declarations 12-5
12.3.2 Elaboration of a Specification 12-5
12.3.2.1 Attribute Specifications 12-5
12.3.2.2 Configuration Specifications 12-6
12.3.2.3 Disconnection Specifications 12-6
12.4 Elaboration of a Statement Part 12-6
12.4.1 Block Statements 12-7
12.4.2 Generate Statements 12-7
12.4.3 Component Instantiation Statements 12-8
12.4.4 Other Concurrent Statements 12-8
12.5 Dynamic Elaboration 12-8
12.6 Execution of a Model 12-9
12.6.1 Propagation of Signal Values 12-9
12.6.2 Updating Implicit Signals 12-12
12.6.3 The Simulation Cycle 12-13

CHAPTER 13 LEXICAL ELEMENTS

13.1 Character Set 13-1
13.2 Lexical Elements, Separators, and Delimiters 13-3
13.3 Identifiers 13-4
13.4 Abstract Literals 13-4
13.4.1 Decimal Literals 13-4
13.4.2 Based Literals 13-5
13.5 Character Literals 13-6
13.6 String Literals 13-6
13.7 Bit String Literals 13-7
13.8 Comments 13-7
13.9 Reserved Words 13-8
13.10 Allowable Replacements of Characters 13-10

CHAPTER 14 PREDEFINED LANGUAGE ENVIRONMENT

14.1 Predefined Attributes 14-1
14.2 Package STANDARD 14-9
14.3 Package TEXTIO 14-11

APPENDIX A SYNTAX SUMMARY

APPENDIX B GLOSSARY

CHAPTER 1

DESIGN ENTITIES AND CONFIGURATIONS

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 sub-system, 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 Section 9.1).

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 of 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 chapter 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*.

1.1 Entity Declarations

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 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_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 must repeat the identifier of the entity declaration.

1.1.1 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 ]

generic_clause ::=
    generic ( generic_list ) ;

port_clause ::=
    port ( port_list ) ;
```

The generic list in the formal generic clause defines generic constants whose values may be determined by the environment. The port list in the formal port clause defines the input/output ports of the design entity.

In certain circumstances, the names of generic constants and ports declared in the entity header become visible outside of the design entity (see Sections 10.2 and 10.3).

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 AndGate ;

-- an entity declaration with neither:

    entity TestBench is
    end TestBench ;
```

1.1.1.1 Generics

Generics provide a channel for static information to be communicated to a block from its environment. The following applies to both external blocks defined by design entities and to internal blocks defined by block statements.

generic_list ::= *generic*_interface_list

The generics of a block are defined by a generic interface list; interface lists are described in Section 4.3.3.1. 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, 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 and no default expression is present in the corresponding interface element.

Note:

Generics may be used to control structural, dataflow, or behavioral characteristics of a block, 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, location, etc.

1.1.1.2 Ports

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.

port_list ::= *port*_interface_list

The ports of a block are defined by a port interface list; interface lists are described in Section 4.3.3.1. Each interface element in the port interface list declares a formal port.

The ports of a block may be associated with signals in the environment in which the block is used, in order to communicate with other blocks in that environment. A port is itself a signal (see Section 4.3.1.2), thus a formal port of a block may be associated with a port of an enclosing block. The port or signal associated with a given formal port is called the *actual* corresponding to the formal port (see Section 4.3.3.2). The actual must be denoted by a static name (see Section 6.1).

If, when a given description is completely elaborated (see Chapter 12), a formal port is associated with an actual that is itself a port, then the following restrictions apply depending upon the mode (see Section 4.3.3) of the formal port:

1. For a formal port of mode **in**,
   the associated actual may only be a port of mode **in**, **inout**, or **buffer**.

2. For a formal port of mode **out**,
   the associated actual may only be a port of mode **out** or **inout**.

3. For a formal port of mode **inout**,
   the associated actual may only be a port of mode **inout**.

4. For a formal port of mode **buffer**,
   the associated actual may only be a port of mode **buffer**.

5. For a formal port of mode **linkage**,
   the associated actual may be a port of any mode.

A **buffer** port may have at most one source (see Section 4.3.1.2). Furthermore, any actual associated with a formal buffer port may have at most one source.

If a formal port is associated with an actual port or signal, 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*. A port of mode **in** may be unconnected only if its declaration includes a default expression (see Section 4.3.3). A port of any mode other than **in** may be unconnected as long as its type is not an unconstrained array type.

1.1.2 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
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | file_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | disconnection_specification
    | use_clause
```

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.

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;
```

1.1.3 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
    | passive_process_statement
```

Only concurrent assertion statements, concurrent procedure call statements, or process statements may appear in the interface statement part. All such statements must be passive (see Section 9.2). 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 := 12ns;
        constant PulseWidth: Time := 50 ns;

        use  Work.TimingMonitors.all;

    begin
        assert Clk='1' or Clk'Delayed'Stable (PulseWidth);
        CheckTiming (Setup, Din, Load, Clk);
    end ;
```

1.2 Architecture Bodies

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_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 must reside in the same library.

If a simple name appears at the end of an architecture body, it must 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.

1.2.1 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
  | type_declaration
  | subtype_declaration
  | constant_declaration
  | signal_declaration
  | file_declaration
  | alias_declaration
  | component_declaration
  | attribute_declaration
  | attribute_specification
  | configuration_specification
  | disconnection_specification
  | use_clause
```

The various kinds of declaration are described in Chapter 4, Declarations, and the various kinds of specification are described in Chapter 5, Specifications. The use clause, which makes externally defined names visible within the block, is described in Chapter 10, Scope and Visibility.

1.2.2 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 Chapter 9, Concurrent Statements.

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 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);

        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 10ns;
        end process;

    end Behavior;
```

1.3 Configuration Declarations

The binding of component instances to design entities is performed by configuration specifications (see Section 5.2); 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
        block_configuration
    end [ configuration_simple_name ] ;

configuration_declarative_part ::=
    { configuration_declarative_item }

configuration_declarative_item ::=
    use_clause
    | attribute_specification
```

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

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

Note:

A configuration declaration achieves its effect entirely through elaboration (see Chapter 12). There are no dynamic semantics associated with a configuration declaration.

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

Example:

```
-- 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 StructureView
            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 V4_27_87 ;
```

1.3.1 Block Configuration

A block configuration defines the configuration of a block. Such a block may be either an internal block defined by a block statement or an external block defined by a design entity.

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

block_specification ::=
    architecture_name
    | block_statement_label
    | generate_statement_label [ ( index_specification ) ]

index_specification ::=
    discrete_range
    | static_expression

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 must be an architecture name, and that architecture name must 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 must be fully bound (see Section 5.2.1.1), the block specification of

that block configuration must be an architecture name, and that architecture name must 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 must be a block statement or generate statement label, and the label must 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 Section 10.2) 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.

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

For any name that is the label of a generate statement within a given block, one or more corresponding block configurations may appear as configuration items 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 an index specification that is a 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 index. If such a block configuration contains an index 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 index. If no index specification appears in such a block configuration, then it applies to all implicit blocks generated by the corresponding generate statement.

Within a given block configuration, 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:

As a result of the rules described above and in Chapter 10, 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).

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

Examples:

```
-- a block configuration for a design entity:

    for Work.ShiftReg                -- an architecture name
        -- configuration items
        -- for blocks and components
        -- within ShiftReg
    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;
```

1.3.2 Component Configuration

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

```
component_configuration ::=
    for component_specification
        [ use binding_indication ; ]
        [ block_configuration ]
    end for;
```

The component specification (see Section 5.2) 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 both an explicit configuration specification (in an architecture body) and a component configuration containing a binding indication (in a configuration declaration) apply to the same component instance.

If the component configuration contains a binding indication (see Section 5.2.1), 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 Section 5.2.2). 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.

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 (StdCells.DataFlow)
            port map (Pout=>A, Pin=>B, IO=>Dir, Vdd=>Pwr, Gnd=>Gnd) ;
    end for;

-- a component configuration containing block configurations:

    for D1: DSP
        -- 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;
```

CHAPTER 2

SUBPROGRAMS AND PACKAGES

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 to share the same declarations.

There are two forms of subprogram: procedures and functions. A procedure call is a statement; a function call is an expression and returns a value. 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.

2.1 Subprogram Declarations

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

```
subprogram_declaration ::=
    subprogram_specification ;

subprogram_specification ::=
    procedure designator [ ( formal_parameter_list ) ]
    | function designator [ ( formal_parameter_list ) ] return type_mark

designator ::= identifier | operator_symbol

operator_symbol ::= string_literal
```

The specification of a procedure specifies its designator and its *formal parameters* (if any). The specification of a function specifies its designator, its formal parameters (if any), and the subtype of the returned value (the *result subtype*). 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. The sequence of characters represented by an operator symbol must be an operator belonging to one of the six

classes of operators defined in Section 7.2. Extra spaces are not allowed in an operator symbol, and the case of letters is not significant.

Note:

All subprograms can be called recursively.

2.1.1 Formal Parameters

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, or signals. In all three cases, the mode of a parameter determines how a given formal parameter may be accessed within the subprogram. The mode of a formal parameter, together with its class, may also determine how such access must be implemented.

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.

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 must be **constant** or **signal**. If no object class is explicitly given, **constant** is assumed.

In a subprogram call, the actual designator associated with a formal parameter of class **signal** must be a signal. The actual designator associated with a formal of class **variable** must be a variable. The actual designator associated with a formal of class **constant** must be an expression.

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, and such references retrieve the value of the attribute associated with the formal.

2.1.1.1 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 section.

For a parameter of a scalar type or an access type, 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. After completion of the subprogram body, if the mode is **inout** or **out**, the value of the formal parameter is copied back into the associated actual parameter.

For a parameter whose type is an array or record, an implementation may pass parameter values by copy, as for scalar types. If a parameter of mode **out** is passed by copy, then the range of each index position of the actual parameter must be copied in, and likewise for its subelements. 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 (variable) parameter whose type is a file type, no particular parameter passing mechanism is defined by the language, but a reference to the formal parameter must 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 type conversion function.

Note:

Within the body of a subprogram, a formal parameter is subject to any constraint resulting from the subtype indication given in the parameter specification. For a formal parameter of an unconstrained array type, the ranges of each index position are obtained from the actual parameter, and the formal parameter is constrained by these ranges.

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 program using such an undefined value is erroneous.

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

Note:

Within the body of a subprogram, a formal signal parameter is subject to any constraint resulting from the subtype indication given in the parameter specification. For a formal signal parameter of an unconstrained array type, the bounds are obtained from the actual signal parameter, and the formal parameter is constrained by these bounds.

It is a consequence of the above 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.

2.2 Subprogram Bodies

A subprogram body specifies the execution of a subprogram.

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

subprogram_declarative_part ::=
    { subprogram_declarative_item }

subprogram_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause

subprogram_statement_part ::=
    { sequential_statement }
```

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 must be a corresponding body. If both a declaration and a body are given, the subprogram specification of the body must conform (see Section 2.7) to the subprogram specification of the declaration. Furthermore, both the declaration and the body must occur immediately within the same declarative region.

If a designator appears at the end of a subprogram body, it must repeat the designator of the subprogram.

The algorithm performed by a subprogram is defined by the sequence of statements that appear in the subprogram statement part.

The execution of a 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. Upon completion of the body, 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 procedure if that process or subprogram contains a procedure call statement for the given procedure or for a parent of the given procedure.

If a function subprogram is a parent of a given procedure, and that procedure contains a reference to a signal or variable object, then that object must be declared within the declarative region associated with the function or within the declarative region associated with the procedure. Similarly, if a function subprogram contains a reference to a signal or variable object, then that object must be declared within the declarative region associated with the function.

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

The above rules concerning function subprograms, together with the fact that function parameters may only be of mode **in**, imply that a function has no effect other than the computation of the returned value. Thus a 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.

2.3 Subprogram Overloading

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 at each parameter position 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 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 in several subprogram specifications. 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 illegal) 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 specification.

Examples: :

```
-- declarations of overloaded subprograms:

    procedure Write (F: inout Text; Value: Integer) ;
    procedure Write (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:

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

    Check (Setup=>10ns, D=>Bus, C=>Clk1) ;
    Check (Hold=>5ns, D=>Bus, C=>Clk2);
    Check (15ns, Bus, Clk) ;   -- ambiguous if Data'Base = Clock'Base
```

Note:

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.

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 Section 6.3).

2.3.1 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 must be one of the operators in the six operator classes defined in Section 7.2.

The subprogram specification of a unary operator must have a single parameter. The subprogram specification of a binary operator must have 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 "-", overloading is allowed both as a unary operator and as a binary operator.

Note:

Overloading of the equality operator does not affect the selection of choices in a case statement or in a selected signal assignment statement.

Overloading a short-circuit operator such as **and** does not imply that the function designated by the operator symbol will be invoked in a short-circuit manner.

Functions that overload operator symbols may also be called using function call notation rather than operator notation.

Examples:

```
type MVL is ('0', '1', 'Z', 'X') ;

function "and" (L,R: MVL) return MVL ;
function "or" (L,R: MVL) return MVL ;
function "not" (R: MVL) return MVL ;

signal Q,R,S: MVL ;

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

2.4 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 Section 4.3.1.2).

A resolution function must have a single input parameter that is a one-dimensional, unconstrained array whose element type is that of the resolved signal. The index subtype of this array must be sufficient for the number of sources of any signal resolved with this function. The type of the return value of the function must also be that of the signal.

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 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.

Resolution functions are implicitly invoked during each simulation cycle in which corresponding resolved signals are active (see Section 12.6.1). 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 Section 8.3.1). 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 must return 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;
```

2.5 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.

```
package_declaration ::=
    package identifier is
        package_declarative_part
    end [ package_simple_name ] ;

package_declarative_part ::=
    { package_declarative_item }

package_declarative_item ::=
    subprogram_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | file_declaration
    | alias_declaration
    | component_declaration
    | attribute_declaration
    | attribute_specification
    | disconnection_specification
    | use_clause
```

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

Items declared within a 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 Section 10.4).

Note:

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

A subprogram written in another language can be made available by defining its interface via a subprogram declaration within a package declaration that has no corresponding package body. The body of such a subprogram must be associated with the declaration of its interface in some implementation-dependent fashion. For example, built-in functions provided by a given simulator might be declared in this manner. Foreign language subprograms declared in this manner are assumed to implement the semantics implied by their interface declarations.

Examples:

```
-- a package declaration that needs no package body:

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

-- a package declaration that may have 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 TriState ;
```

2.6 Package Bodies

A package body defines the bodies of subprograms declared in the interface to the package or 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_simple_name ] ;

package_body_declarative_part ::=
    { package_body_declarative_item }

package_body_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | file_declaration
    | alias_declaration
    | use_clause
```

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

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 Section 4.3.1.1), then a constant declaration with the same identifier must 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 must 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 TriState ;
```

2.7 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 selector, 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 *conform* if, apart from comments and the above allowed 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.

Conformance is likewise defined for subtype indications in deferred constant declarations.

Note:

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.

The following specifications do not 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)

CHAPTER 3

TYPES

This chapter 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 Chapter 14.

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 predefined operators (see Section 7.2). These operations are each implicitly declared for a given type declaration, immediately after the type declaration and before the next explicit declaration, if any.

There are four 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.

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.

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 manual 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 must not have a subelement whose type is the given type itself.

The name of a class of types is used in this manual 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:

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.

3.1 Scalar Types

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 which 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. The smaller of L and R is called the *lower bound*, and the larger, the *upper bound*, of the range. The value V is said to *belong to the range* if the relations (*lower bound* <= V) and (V <= *upper bound*) are both true and the range is not a null range. The operators >, <, and <= in the above definitions are the predefined operators of the applicable scalar type.

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) must 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.

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.

3.1.1 Enumeration Types

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 must be distinct. Each enumeration literal specification is the declaration of the corresponding enumeration literal.

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 must be determinable from the context.

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'
```

3.1.1.1 Predefined Enumeration Types

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

The predefined type CHARACTER is a character type whose values are the 128 characters of the *ASCII* character set. Each of the 95 graphic characters of this character set is denoted by the corresponding character literal.

The declarations of the predefined types CHARACTER, BIT, BOOLEAN, and SEVERITY_LEVEL appear in package STANDARD in Chapter 14.

Note:

The non-graphic elements of the predefined type CHARACTER are the ASCII abbreviations for the non-printing characters in the ASCII set (except for those noted in Chapter 14).

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.

3.1.2 Integer Types

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 must 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 must 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 manual. 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 Section 7.3.5).

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 Section 7.2). 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 must 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;

3.1.2.1 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 may be determined from the 'LOW and 'HIGH attributes.

3.1.3 Physical Types

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

```
physical_type_definition ::=
    range_constraint
        units
            base_unit_declaration
            { secondary_unit_declaration }
        end units

base_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 must 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 must 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 base unit declaration or a secondary unit declaration) defines a *unit name*. Unit names declared in secondary unit declarations must be directly or indirectly defined in terms of integral multiples of the base unit of the type declaration in which they appear.

The abstract literal portion (if present) of a physical literal appearing in a secondary unit declaration must 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 base 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 operations are predefined for all physical types (see Section 7.2). It is an error 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 physical type).

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

Examples:

```
type TIME 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
    -- base unit:
        A;                      -- angstrom
    -- metric lengths:
        nm    = 10 A;           -- 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 A;       -- 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;

x: distance;  y: time;  z: integer;

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

Note:

The above definitions imply that, if 1 is not in the range specified by the physical type definition, then the name of the base unit standing alone is not a legal literal of the physical type.

The POS and VAL attributes may be used to convert between abstract values and physical values.

3.1.3.1 Predefined Physical Types

The only predefined physical is type TIME. The range of TIME is implementation-dependent, but it is guaranteed to include the range -2147483647 to +2147483467. It is defined with an ascending range. All specifications of delays must be of type TIME. The declaration of type TIME appears in package STANDARD in Chapter 14.

By default, the base unit of type TIME (1 femtosecond) is the *resolution limit* for type TIME. Any TIME values smaller than this limit are truncated to zero (0) time units. An implementation may allow a given execution of a model (see Section 12.6) 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 executed, and the position number of that unit is less than that of the secondary unit selected as the resolution limit for type TIME during the execution of the model.

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.

3.1.4 Floating Point Types

Floating point types provide approximations to the real numbers. Floating point types are useful for models in which the precise characterization of a floating point calculation is not important or not determined.

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 must 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 must 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 manual. 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 Section 7.3.5).

The same arithmetic operations are predefined for all floating point types (see Section 7.2). It is an error 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). However, in the case of operations on floating point types, an implementation is not required to detect such an error, since the detection of overflow conditions resulting from floating point operations is not easily accomplished on many hosts.

An implementation may restrict the bounds of the range constraint of floating point types, other than type *universal_real*. However, an implementation must allow the declaration of any floating point type whose range is wholly contained within the bounds -1E38 and +1E38, inclusive. The representation of floating point types must include a minimum of six decimal digits of precision.

3.1.4.1 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 include the range -1E38 to +1E38. It is defined with an ascending range.

Note:

The range of REAL in a particular implementation may be determined from the 'LOW and 'HIGH attributes.

3.2 Composite Types

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. A composite type may only contain elements that are of scalar, composite, or access types; elements of file types are not allowed in a composite type. Thus an object of a composite type ultimately represents a collection of objects of scalar or access types, one for each non-composite subelement of the composite object.

3.2.1 Array Types

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 ::=
    unconstrained_array_definition | constrained_array_definition

unconstrained_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 <>

index_constraint ::= ( discrete_range { , discrete_range } )

discrete_range ::= discrete_subtype_indication | range
```

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 unconstrained array definition 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 must 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).

A constrained array definition 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) unconstrained array definition, in which the element subtype indication is that of the constrained array definition, and 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.

If a constrained array definition 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 must not contain a resolution function.

Examples:

```
-- Examples of 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 unconstrained array declarations

type MEMORY is array (INTEGER range <>) of MY_WORD ;
    -- a memory array type

-- 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
```

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 to MAX) of ELEMENT;
```

is equivalent to the sequence of declarations

```
subtype index_subtype is POSITIVE range MIN to MAX;
type array_type is array (index_subtype range <>) of ELEMENT;
subtype T is array_type (index_subtype);
```

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

3.2.1.1 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 each bound is either a numeric literal or an attribute, and the type of both bounds (prior to the implicit conversion) is

the type *universal_integer*. Otherwise, both bounds must be of the same discrete type, other than *universal_integer*; this type must be determined independently of the context, but using the fact that the type must be discrete and that both bounds must have the same type. These rules apply also to a discrete range used in an iteration scheme or a generation scheme.

If an index constraint appears after a type mark in a subtype indication, then the type or subtype denoted by the type mark must not already impose an index constraint. The type mark must denote either an unconstrained array type, or an access type whose designated type is such an array type. In either case, the index constraint must provide a discrete range for each index of the array type, and the type of each discrete range must be the same as that of the corresponding index.

An index constraint is *compatible* with the type denoted by the type mark if and only if the constraint defined by each discrete range is compatible with the corresponding index subtype. If any of the discrete ranges defines a null range, any array thus constrained is a *null array*, having no components. 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 type conversion.)

The index range for each index of an array object is determined as follows:

- For a variable or signal declared by an object declaration, the subtype indication of the corresponding object declaration must define a constrained array subtype (and thereby, the index range for each index of the object). The same requirement exists for the subtype indication of an element declaration, if the type of the record element is an array type; and for the element subtype indication of an array type definition, if the type of the array element is itself an array type.

- For a constant declared by an object declaration, the index ranges are defined by the initial value, if the subtype of the constant is unconstrained; otherwise, they are defined by this subtype (in which case the initial value is the result of an implicit subtype conversion).

- For an attribute whose value is specified by an attribute specification, the index ranges are defined by the expression given in the specification, if the subtype of the attribute is unconstrained; otherwise, they are defined by this subtype (in which case the value of the attribute is the result of an implicit subtype conversion).

- For an array object designated by an access value, the index ranges are defined by the allocator that creates the array object (see Section 7.3.6).

- For an interface object declared with a subtype indication that defines a constrained array subtype, the index ranges are defined by that subtype.

- For a formal parameter of a subprogram that is of an unconstrained array type, the index ranges are obtained from the corresponding association element in the applicable subprogram call.

- For a formal generic of a design entity that is of an unconstrained array type, the index ranges are obtained from the corresponding association element in the generic map clause of the applicable (implicit or explicit) binding indication.

- For a formal port of a design entity that is of an unconstrained array type, the index ranges are obtained from the corresponding association element in the port map clause of the applicable (implicit or explicit) binding indication.

- For a local generic of a component that is of an unconstrained array type, the index ranges are obtained from the corresponding association element in the generic map clause of the applicable component instantiation statement.

- For a local port of a component that is of an unconstrained array type, the index ranges are obtained from the corresponding association element in the port map clause of the applicable component instantiation statement.

If the index ranges for an interface object are obtained from the corresponding association element, then they are determined either by the actual part or the formal part of the association element, depending upon the mode of the interface object, as follows:

- For an interface object of mode **in**, **inout**, or **linkage**, if the actual part includes a type conversion function, then the result type of that function must be a constrained array subtype, and the index ranges are obtained from this constrained subtype; otherwise, the index ranges are obtained from the object or value denoted by the actual designator.

- For an interface object of mode **out**, **buffer**, **inout**, or **linkage**, if the formal part includes a type conversion function, then the parameter subtype of that function must be a constrained array subtype, and the index ranges are obtained from this constrained subtype; otherwise, the index ranges are obtained from the object denoted by the actual designator.

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

3.2.1.2 Predefined Array Types

The predefined array types are STRING and BIT_VECTOR, defined in package STANDARD in Chapter 14.

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 type BIT_VECTOR are one-dimensional arrays of the predefined type BIT, indexed by values of the predefined subtype NATURAL:

```
subtype NATURAL is INTEGER range 0 to INTEGER'HIGH ;
type BIT_VECTOR is array (NATURAL range <>) of BIT ;
```

Examples:

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

signal LOW_BYTE : BIT_VECTOR (0 to 7) ;
```

3.2.2 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

element_declaration ::=
    identifier_list : element_subtype_definition ;

identifier_list ::= identifier { , identifier }

element_subtype_definition ::= subtype_indication
```

Each element declaration declares an element of the record type. The identifiers of all elements of a record type must 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.

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.

Example:

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

3.3 Access Types

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 Section 7.3.6) 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** which 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*; the base type of this subtype is called the *designated type*. The designated type must not be a file type.

An object declared to be of an access type must 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 index 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** BUFFER;

Note:

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.

If the type of the object designated by the access value is an array type, this object is constrained with the array bounds supplied implicitly or explicitly for the corresponding allocator.

3.3.1 Incomplete Type Declarations

There are no particular limitations on the designated type of an access type. 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 must be a corresponding full type declaration with the same identifier. This full type declaration must 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;                -- incomplete type declaration
type LINK is access CELL;
type CELL is
    record
        VALUE  : INTEGER;
        SUCC   : LINK;
        PRED   : LINK;
    end record;
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;
        CONNECTIONS   : WIRE_LIST_PTR;
    end record;

type WIRE is
    record
        WIRE_NAME     : STRING;
        CONNECTS      : PART_LIST_PTR;
    end record;
```

3.3.2 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 Section 7.3.6. 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 explicitly deallocate 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.

3.4 File Types

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 constrained or an unconstrained subtype. The base type of this subtype must not be a file type or an access type. If the base type is a composite type, it must not contain a subelement of an access type. If the base type is an array type, it must be a one-dimensional array type.

Examples:

```
file of STRING     -- Defines a file type that can contain
                   -- an indefinite number of strings
file of NATURAL    -- Defines a file type that can contain
                   -- only non-negative integer values
```

3.4.1 File Operations

Three operations are provided 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 constrained array subtype, the following operations are implicitly declared immediately following the file type declaration:

procedure READ (F: **in** FT; VALUE: **out** TM);

procedure WRITE (F: **out** FT; VALUE: **in** TM);

function ENDFILE (F: **in** FT) **return** BOOLEAN;

Procedure READ retrieves the next value from a file. Procedure WRITE appends a value to a file. Function ENDFILE returns False if a subsequent READ operation on an input file can

retrieve another value from the file; otherwise it returns True. Function ENDFILE always returns True for an output file.

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

procedure READ (F: **in** 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.

Note:

Predefined package TEXTIO is provided to support formatted ASCII I/O. It defines type TEXT (a file type representing files of variable-length ASCII 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 Chapter 14.

CHAPTER 4

DECLARATIONS

The language defines several kinds of entities that are declared explicitly or implicitly by declarations.

```
declaration ::=
    type_declaration
  | subtype_declaration
  | object_declaration
  | file_declaration
  | interface_declaration
  | alias_declaration
  | attribute_declaration
  | component_declaration
  | entity_declaration
  | configuration_declaration
  | subprogram_declaration
  | package_declaration
```

For each form of declaration the language rules define a certain region of text called the *scope* of the declaration. Each form of declaration associates an identifier with a declared entity. Only within its scope, there are places where it is possible to use the identifier to refer to the associated declared entity; these places are defined by the visibility rules. At such places the identifier is said to be a *name* of the entity; the name is said to *denote* the associated entity.

This chapter describes type and subtype declarations, the various kinds of object declaration, alias declarations, attribute declarations, and component declarations. The other kinds of declarations are described in Chapter 1, Design Entities and Configurations, and in Chapter 2, Subprograms and Packages.

A declaration takes effect through the process of elaboration. Elaboration of declarations is discussed in Chapter 12.

4.1 Type Declarations

A type declaration declares a 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

The types created by the elaboration of distinct type definitions are distinct types. The elaboration of the type definition for a scalar type or a constrained array 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 reference manual 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:

Two type definitions always define two distinct types, even if they are textually 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.

The various forms of type definition are described in Chapter 3, Types. Examples of type declarations are also given in that chapter.

4.2 Subtype Declarations

A subtype declaration declares a subtype.

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

subtype_indication ::=
[*resolution_function*_name] type_mark [constraint]

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

```
constraint ::=
    range_constraint
  | index_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.

If a subtype indication includes a resolution function name, then any signal declared to be of that subtype will be resolved, if necessary, by the named function (see Section 2.4). A resolution function name 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 section. 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 or a file type may not contain a resolution function. Furthermore, the only allowable constraint on a subtype indication denoting an access type is an index constraint (and then only if the designated type is an array type).

Note:

A subtype declaration does not define a new type.

4.3 Objects

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

- an object declared by an object declaration.
- a file declared by a file declaration,
- a loop or generate index.
- a formal parameter of a subprogram.
- a formal port of a design entity.

- a formal generic.
- a local port.
- a local generic.
- an element or slice of another object.
- an object value designated by a value of an access type.

There are three classes of objects: constants, signals, and variables. The class of an explicitly declared object is specified by the reserved word that must 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 as the given object. The value of a composite object is the aggregation of the values of its subelements.

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

4.3.1 Object Declarations

An object declaration declares an object of a specified type.

```
object_declaration ::=
    constant_declaration
  | signal_declaration
  | variable_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 Section 4.3.3).

4.3.1.1 Constant Declaration

A constant declaration declares a *constant* of the specified type.

```
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 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*. Such a constant declaration may only appear in a package declaration. The corresponding full constant declaration, which defines the value of the constant, must appear in the body of the package (see Section 2.6).

Formal parameters of subprograms that are of mode **in** may be constants, and local and formal generics are always constants; the declarations of such objects are discussed in Section 4.3.3. A loop index is a constant within the corresponding loop; similarly a generate index is a constant within the corresponding generate statement; 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 or an access type.

Examples:

```
constant TOLERANCE : DISTANCE := 1.5nm ;
constant PI : REAL := 3.141592 ;
constant CYCLE_TIME : TIME := 100ns ;
constant Propagation_Delay ;
```

4.3.1.2 Signal Declaration

A signal declaration declares a *signal* of the specified type.

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

signal_kind ::= register | bus
```

If the name of a resolution function appears in the declaration of a signal, or in the declaration of the subtype used to declare the signal, then that resolution function is associated with the declared signal. Such a signal 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. A guarded signal is assigned values under the control of Boolean-valued *guard expressions* (or *guards*). When a given guard becomes False, the drivers of the corresponding guarded signals are implicitly assigned a null transaction (see Section 8.3.1) to cause those drivers to turn off. A disconnection specification (see Section 5.3) 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 must 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 or 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 Section 9.2.1) or an **out, inout, buffer,** or **linkage** port of a component instance 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.

If a subelement of a resolved signal of composite type is associated as an actual in a port map clause (either in a component instantiation statement or in a binding indication), and the corresponding formal is of mode **out, inout, buffer,** or **linkage**, then every scalar subelement of that signal must be associated exactly once with such a formal in the same port map clause, 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 clause, that is equivalent to each of its subelements being associated in the same port map clause.

If a subelement of a resolved signal of composite type has a driver in a given process, then every scalar subelement of that signal must 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 MultiValuedLogic ;
```

Note:

Ports of any mode are also signals. The term *signal* is used in this manual to refer to objects declared either by signal declarations or by port declarations; the term *port* is used to refer to objects declared by port declarations only.

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 Section 12.6.3).

The value of a signal may be indirectly modified by a signal assignment statement (see Section 8.3); such assignments affect the future values of the signal.

4.3.1.3 Variable Declaration

A variable declaration declares a *variable* of the specified type.

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

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 must be that of the variable. Such an expression is said to be an *initial value expression.*

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.

It is an error if a variable declaration declares a variable that is of a file type.

Note:

The value of a variable may be modified by a variable assignment statement (see Section 8.4); such assignments take effect immediately. Procedure parameters of mode **in** may be file variables; procedure parameters of mode **out** or **inout** may be variables of any kind.

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 may therefore persist 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.

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, or 1.

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

4.3.2 File Declarations

A *file object* is created by a file declaration. Such an object is a member of the variable class of objects; however, the operations on file objects are restricted in comparison to those available on other variable objects. In particular, assignment to a file object is not allowed.

file_declaration ::=
file identifier : subtype_indication **is** [mode] file_logical_name ;

file_logical_name ::= *string*_expression

The subtype indication of a file declaration must define a file subtype. The only modes allowed in an external file association are **in** and **out**.

The file logical name must 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 must 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 the mode specified in the file declaration is the mode **in**, then the contents of the external file may be read by processes during simulation. In this case, the file object may be read, but not updated, by one or more processes. If the mode specified in the file declaration is the mode **out**, then the contents of the external file may be written by processes during simulation. In this case, the file object may be updated, but not read, by one or more processes. The default mode is **in**, if no mode is specified.

If multiple file objects are associated with the same file logical name, and each file object is declared with a file declaration that specifies mode **out**, then values written to each file object are written to an external file identified by that file logical name. The language does not define the order in which such values are written to the external file, nor does it define whether one external file or multiple external files are created as a result.

If a formal subprogram parameter is of a file type, it must be associated with an actual that is a file object. A file object of a given mode may only be passed to a formal file variable of the corresponding mode.

Note:

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

4.3.3 Interface Declarations

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

interface_declaration ::=
interface_constant_declaration
| interface_signal_declaration
| interface_variable_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]

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

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

For an interface constant declaration or an interface signal declaration, the subtype indication must define a subtype that is neither a file type nor an access 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 must be that of the corresponding interface object. It is an error if a default expression appears in an interface declaration and the mode is **linkage** or the corresponding type mark denotes a file type.

In an interface signal declaration, 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 Section 4.3.1.2). The implicit value is used to determine the initial contents of drivers of the interface signal, if any, 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 Section 4.3.3.2.

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.

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

- When it appears as the target of an assignment statement, and also (indirectly) when the object is associated with an interface object of the modes **out, buffer, inout**, or **linkage**.
- When one of its subelements is updated.
- When the object is a file and a WRITE operation is performed on the file.

Only signal or variable objects may be updated. A variable of a file type can be updated only by performing a WRITE operation; it is an error if a file variable appears as the target of an assignment statement.

An interface object has one of the following modes:

- **in**. The value of the interface object may only be read. In addition, any attributes of the interface object may be read, except that attributes STABLE, QUIET, DELAYED, and TRANSACTION of a signal parameter may not be read within a subprogram. For a file object, operation ENDFILE is allowed.
- **out**. The value of the interface object may be updated. Reading the attributes of the interface element, other than the predefined attributes STABLE, QUIET, DELAYED, TRANSACTION, EVENT, ACTIVE, LAST_EVENT, LAST_ACTIVE, and LAST_VALUE, is allowed. For a file object, operation ENDFILE is allowed. No other reading is allowed.
- **inout**. The value of the interface object may be both read and updated. Reading the attributes of the interface object is also permitted. For a file object, operation ENDFILE is allowed.
- **buffer**. The value of the interface object may be both read and updated. Reading the attributes of the interface object is also permitted.
- **linkage**. The value of the interface object may be read or updated, but only by appearing as an actual corresponding to an interface object of mode **linkage**. No other reading or updating is permitted.

Note:

Although signals of modes **inout** and **buffer** have the same characteristics with respect to whether they may be read or updated, a signal of mode **inout** may be updated by zero or more sources, whereas a signal of mode **buffer** must be updated by at most one source (see Section 1.1.1.2).

A subprogram parameter that is of a file type must be declared as a variable parameter.

4.3.3.1 Interface Lists

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

```
interface_list ::=
    interface_element { ; interface_element }

interface_element ::=  interface_declaration
```

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

4.3.3.2 Association Lists

An association list establishes correspondences between formal or local generic, port, or parameter names on the one hand and local or actual names or expressions on the other.

```
association_list ::=
    association_element { , association_element }

association_element ::=
    [ formal_part => ] actual_part

formal_part ::=
    formal_designator
    | function_name ( formal_designator )

formal_designator ::=
    generic_name
    | port_name
    | parameter_name

actual_part ::=
    actual_designator
    | function_name ( actual_designator )

actual_designator ::=
    expression
    | signal_name
    | variable_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, or block statement. 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 must occur first, at their

normal position. Hence once a named association is used, the rest of the association list must use only named associations.

In the following, the term *actual* refers to an actual designator, and the term *formal* refers to a formal designator.

The formal part of a named element association may be in the form of a function call, where the single argument of the function is the formal designator itself. In this case, the function name must 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 function provides for type conversion in the event that data flows from the formal to the actual.

Similarly, the actual part of a (named or positional) element association may be in the form of a function call, where the single argument of the function is the actual designator itself. In this case, the function name must denote a function whose single parameter is of the type of the actual, and whose result is the type of the corresponding formal. Such a function provides for type conversion in the event that data flows from the actual to the formal.

If the mode of the formal is **in**, **inout**, or **linkage**, and the actual is not **open**, then the type of the actual (after applying the type conversion function, if present in the actual part) must be the same as the type of the corresponding formal. Similarly, if the mode of the formal is **out**, **inout**, **buffer**, or **linkage**, and the actual is not **open**, then the type of the formal (after applying the type conversion function, if present in the formal part) must 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 type conversion function is present in either the actual part or the formal part of the association element. If a type conversion function 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 Section 2.1.1.1).

A formal may either be an explicitly declared interface object or it may be a subelement of such an interface object. In the latter case, named association must be used to associate the formal and actual. Furthermore, every subelement of the explicitly declared interface object must be associated exactly once with an actual in the same association list, and all such associations must appear in a contiguous sequence within that association list. Each such association element must identify the formal with a locally static name.

If an interface element in an interface list includes a default expression for a formal generic, or for a formal parameter of mode **in**, then any corresponding association list need not include an association element for that interface element. If the association element is not included in the association list, then the value of the default expression is used as the actual expression in an implicit association element for that interface element.

Note:

It is a consequence of the above 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 must be named associations.

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, since ports must be associated with signals as opposed to values. 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 Section 12.6.1).

4.3.4 Alias Declaration

An alias declaration declares an alternate name for an existing object.

```
alias_declaration ::=
    alias identifier : subtype_indication is name ;
```

The identifier specified in an alias declaration denotes the object represented by the name 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.

The name must be a static name (see Section 6.1) that denotes an object. The base type of the name specified in an alias declaration must be the same as the base type of the type mark in the subtype indication; this type must not be a multi-dimensional array type. When the object denoted by the name is referenced via the alias defined by the alias declaration, it is viewed as if it were of the subtype specified by the subtype indication. The same applies to attribute references where the prefix of the attribute denotes the alias. If this subtype is a one-dimensional array subtype, then the subtype must include a matching element (see Section 7.2.2) for each element of that object denoted by the name.

A reference to an element of an alias is implicitly a reference to the matching element of the object denoted by the alias. A reference to a slice of an 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.

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.

4.4 Attribute Declarations

An attribute is a value, function, type, range, signal, or constant that may be associated with one or more entities in a description. There are two categories of attributes: predefined attributes and user-defined attributes. Predefined attributes provide information about entities in a description. Chapter 14 contains the definition of all predefined attributes. Predefined attributes that are signals may 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, an architecture, a configuration, a procedure, a function, a package, a type, a subtype, a constant, a signal, a variable, a component, or a label.

The type mark must denote a subtype that is neither an access type nor a file type. The subtype need not be constrained.

Examples:

type COORDINATE **is record** X,Y: INTEGER **end record**;
type POSITIVE **is** INTEGER **range** 1 **to** INTEGER'HIGH;
attribute LOCATION: COORDINATE;
attribute PIN_NO: POSITIVE;

Note:

A given entity E will inherit attribute A if and only if an attribute specification for the value of attribute A accompanies the declaration of E. In the absence of such a specification, an attribute name of the form E'A is illegal.

A user-defined attribute is associated with the 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.

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.

4.5 Component Declarations

A component declaration defines a virtual design entity interface 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 design entity that resides in a library.

component_declaration ::=
    component identifier
        [*local*_generic_clause]
        [*local*_port_clause]
    end component ;

Each interface element in the local generic clause declares a local generic. Each interface element in the local port clause declares a local port.

CHAPTER 5

SPECIFICATIONS

This chapter describes *specifications*, which may be used to associate additional information with a VHDL description. A specification associates additional information with a previously declared entity. There are three kinds of specifications: attribute specifications, configuration specifications, and disconnection specifications.

A specification always relates to entities that already exist; thus a given specification must either follow or (in certain cases) be contained within the declaration of the entity to which it relates. Furthermore, a specification must always appear either immediately within the same declarative region as that in which the declaration of the related entity appears, or (in the case of specifications that relate to design units) immediately within the declarative region associated with the declaration of the related entity.

5.1 Attribute Specification

An attribute specification associates a user-defined attribute with one or more entities and defines the value of that attribute for those entities.

```
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

entity_name_list ::=
    entity_designator { , entity_designator }
    | others
    | all

entity_designator ::=  simple_name | operator_symbol
```

The attribute designator must denote an attribute. The entity name list identifies those entities that inherit the attribute, as defined below:

- If a list of entity designators is supplied, then the attribute specification applies to the entities denoted by those designators. It is an error if the class of those names is not the same as that denoted by the entity class.

- If the reserved word **others** is supplied, then the attribute specification applies to entities of the specified class that are declared in the immediately enclosing declarative region, provided that each such entity is not explicitly named in the entity name list of a previous attribute specification.

- If the reserved word **all** is supplied, then the attribute specification applies to all entities of the specified class that are declared in the immediately enclosing declarative region.

An attribute specification with the entity name list **others** or **all** for a given entity class that appears in a declarative region must be the last such specification for the given entity class in that declarative region. No entity in the specified entity class may be declared in a given declarative region following such an attribute specification.

The expression specifies the value of this attribute for each of the entities inheriting the attribute as a result of this attribute specification. The type of the expression in the attribute specification must be the same as (or implicitly convertible to) the type mark in the corresponding attribute declaration.

An attribute specification for an attribute of a design unit (i.e., an entity, an architecture, a configuration, or a package) must appear immediately within the declarative part of that design unit. An attribute specification for an attribute of a procedure, a function, a type, a subtype, an object (i.e., a constant, a signal, or a variable), a component, or a labeled entity must appear within the declarative part in which that procedure, function, type, subtype, object, component, or label, respectively, is declared.

For a given 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 entity. Similarly, it is an error if two different attributes with the same simple name are both associated with a given entity.

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 15pF;
```

Note:

An entity designator that is an operator symbol is used to associate an attribute with an overloaded operator.

If an attribute specification appears, it must follow the declaration of the entity with which the attribute is associated, and it must precede all references to that attribute of that entity. Attribute specifications are allowed for all user-defined attributes, but are not allowed for predefined attributes.

An attribute specification may reference an 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 prefix of the attribute name.

An attribute specification for an attribute of a variable of an access type associates the attribute with the variable itself, not with the designated object. An attribute specification for one of several overloaded subprograms, all of which are declared within the same declarative region, has the effect of associating that attribute with each of the overloaded subprograms.

User-defined attributes represent local information only and cannot be used to pass information from one description to another. For instance, both a signal X within an architectural body and a port Y of a component within that architectural body may have the same attribute A. However, the values of X'A and Y'A are not related in any way. In particular, associating signal X with port Y in a component instantiation statement neither imports the the value Y'A nor exports the value X'A.

5.2 Configuration Specification

A configuration specification associates binding information with component labels representing instances of a given component.

```
configuration_specification ::=
    for component_specification use binding_indication ;

component_specification ::=
    instantiation_list : component_name

instantiation_list ::=
    instantiation_label { , instantiation_label }
    | others
    | all
```

The instantiation list identifies those entities with which binding information is to be associated, as defined below:

- If a list of instantiation labels is supplied, then the configuration specification applies to the corresponding component instances. Such labels must be declared within the immediately enclosing declarative region. It is an error if these component instances are not instances of the component named in the component specification.

- If the reserved word **others** is supplied, then the configuration specification applies to instances of the specified component whose labels are declared in the immediately enclosing declarative region, provided that each such component instance is not explicitly named in the instantiation list of a previous configuration specification.

- If the reserved word **all** is supplied, then the configuration specification applies to all instances of the specified component whose labels are declared in the immediately enclosing declarative region.

A configuration specification with the instantiation list **others** or **all** for a given component name that appears in a declarative region must be the last such specification for the given component name in that declarative region.

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 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.

5.2.1 Binding Indication

A binding indication associates component instances with a particular design entity. It may also associate actuals with formals in the entity interface.

```
binding_indication ::=
    entity_aspect
    [ generic_map_aspect ]
    [ port_map_aspect ]
```

The entity aspect of a binding indication identifies the design entity with which the component instances are associated. If present, the generic map aspect of a binding indication identifies the expressions to be associated with formal generics in the design entity interface. Similarly, the port map aspect of a binding indication identifies the signals to be associated with formal ports in the design entity interface.

If the generic map aspect or port map aspect of a binding indication is not present, then the default rules as described in Section 5.2.2 apply.

5.2.1.1 Entity Aspect

An entity aspect identifies a particular design entity to be associated with component instances. An entity aspect may also specify that such 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* any design entity whose interface is defined by the entity declaration denoted by the entity name. 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 this case, the corresponding component instances are said to be *fully bound.*

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 apex of the design hierarchy that is defined by the configuration denoted by the configuration name.

The third form of entity aspect is used to specify that 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 must not include a generic map aspect or a port map aspect.

If an architecture identifier appears in the entity aspect of a binding indication in a component configuration, then that identifier must be the same as the simple name of an architecture body associated with the entity declaration denoted by the corresponding entity name.

5.2.1.2 Generic Map And Port Map Aspects

A generic map aspect associates values with the formal generics of a block. Similarly, a port map aspect associates signals with the formal ports of block. The following applies to both external blocks defined by design entities and to internal blocks defined by block statements.

generic_map_aspect ::=
    generic map (*generic*_association_list)

port_map_aspect ::=
    port map (*port*_association_list)

Both named and positional association are allowed in a port or generic association list.

The following definitions hold in what follows:

- The term *actual* refers to both an actual designator that appears in an association element of a port association list and an actual designator that appears in an association element of a generic association list.

- The term *formal* refers to both a formal designator that appears in an association element of a port association list and a formal designator that appears in an association element of a generic association list.

The purpose of port and generic map aspects is to associate actuals with the formals of the design entity interface implied by the immediately enclosing binding indication. Each local port or generic of the component instances to which the enclosing configuration specification applies must be associated as an actual with at least one formal. No formal may be associated with more than one actual.

An actual associated with a formal generic in a generic map aspect must be an expression; an actual associated with a formal port in a port map aspect must be a signal.

Certain restrictions apply to the actual associated with a formal port in a port map aspect; these restrictions are described in Section 1.1.1.2.

A formal that is not associated with an actual is said to be an *unassociated* formal.

Note:

A local generic (from a component declaration) or formal generic (from a block statement or from the entity declaration of the enclosing design entity) may appear as an actual in a generic map aspect. Similarly, 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.

5.2.2 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**. Otherwise, if such an entity declaration is visible but has no associated architecture body, then the default entity aspect is of the form

entity *entity*_name

where the entity name is the simple name of the instantiated component. 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.

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 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.

5.3 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
```

The guarded signal specification contains a signal list that identifies the signals for which the implicit disconnection delay is to be defined, as follows:

- If a list of signal names is supplied, then each signal name must be a locally static name that denotes a guarded signal, and the disconnection specification applies to the named signals. Such signals must be declared in the immediately enclosing declarative region.

- If the reserved word **others** is supplied, then the disconnection specification applies to drivers of any signal of the specified type that are declared in the immediately enclosing declarative region, provided that each such signal is not explicitly named in the signal list of a previous disconnection specification.

- If the reserved word **all** is supplied, then the disconnection specification applies to drivers of all signals of the specified type declared in the immediately enclosing declarative region.

A disconnection specification with the signal list **others** or **all** for a given type that appears in a declarative region must be the last such specification for the given type in that declarative region. No guarded signal may be declared in a given declarative region following such a disconnection specification.

The time expression in a disconnection specification must be static and must evaluate to a non-negative value.

It is an error if more than one disconnection specification applies to drivers of the same signal.

In the absence of a disconnection specification for a given scalar signal S of type T, the following default disconnection specification is implicitly assumed:

```
disconnect S : T after 0ns;
```

Thus the implicit disconnection delay for any guarded signal is always defined, either by an explicit disconnection specification or by an implicit one.

CHAPTER 6

NAMES

The rules applicable to the various forms of name are described in this chapter.

6.1 Names

Names can denote declared entities, whether declared explicitly or implicitly. Names can also denote objects denoted by access values, and subelements or slices of composite objects and values. Finally, names can denote attributes of any of the foregoing.

```
name ::=
    simple_name
    | operator_symbol
    | selected_name
    | indexed_name
    | slice_name
    | attribute_name

prefix ::=
    name
    | function_call
```

Certain forms of name (indexed and selected names, slices, 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 Section 7.3.3.

If the type of a prefix is an access type, then the prefix must not be a name that denotes a formal parameter of mode **out**, or a subelement thereof.

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 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.

A name is said to be a *static name* if every expression that appears as part of the name (for example, as an index expression) is a static expression. Furthermore, a name is said to be a *locally static name* if every expression that appears as part of the name is a locally static expression. 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.

Examples:

```
S(C,2)          -- a static name: C is a static constant
R(J to 16)      -- a non-static 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
```

6.2 Simple Names

A simple name for an 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, 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 entity denoted by the name.

6.3 Selected Names

A selected name is used to denote an entity whose declaration appears either within the declaration of another entity or within a design library.

```
selected_name ::= prefix . suffix

suffix ::=
    simple_name
  | character_literal
  | operator_symbol
  | all
```

A selected name may be used to denote an element of a record, an object designated by an access value, or an entity whose declaration is contained within another named entity, particularly within a library or a package. Furthermore, a selected name may be used to denote all 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 must be a simple name denoting an element of a record object or value. The prefix must 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 must be the reserved word **all**. The prefix must belong to an access type.

The remaining forms of selected name are called *expanded names*. The prefix of an expanded name may 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 an entity declared in a package if the prefix denotes the package and the suffix is the simple name, character literal, or operator symbol of an entity whose declaration occurs immediately within that package. An expanded name denotes all entities declared in a package if the prefix denotes the package and the suffix is the reserved word **all**.

An expanded name denotes an entity declared immediately within a named construct if the prefix denotes a construct that is an entity, an architecture, a subprogram, a block statement, a process statement, or a loop statement, and the suffix is the simple name, character literal, or operator symbol of an entity whose declaration occurs immediately within that construct. This form of expanded name is only allowed within the construct itself.

Examples:

```
INSTRUCTION.OPCODE      -- a record element
PTR.all                 -- the object designated by PTR

TTL.SN74LS221           -- a design unit contained in a library
CMOS.all                -- all design units contained in a library

MEASUREMENTS.VOLTAGE    -- an entity declared in a package
STANDARD.all            -- all entities declared in a package

P.DATA                  -- an entity declared in process P
```

6.4 Indexed Names

An indexed name denotes an element of an array.

```
indexed_name ::= prefix ( expression { , expression } )
```

The prefix of an indexed name must be appropriate for an array type. The expressions specify the index values for the element; there must be one such expression for each index position of the array, and each expression must 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
```

6.5 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 must 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 must be of the type of the index of the array. The slice is a *null slice* if the discrete range is a null range, or if the direction of the discrete range is not the same as that of the object 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(24 to 25)       -- a null slice
```

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.

6.6 Attribute Names

An attribute name denotes a value, a function, a type, range, a signal, or a constant associated with an entity.

attribute_name ::=
    prefix ' attribute_designator [(*static*_expression)]

attribute_designator ::= *attribute*_simple_name

The applicable attribute designators depend on the prefix. 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.

If the attribute designator denotes a predefined attribute, the static expression either must or may appear, depending upon the definition of that attribute (see Chapter 14); otherwise, it must not be present.

Examples:

```
REGISTER'LEFT(1)        -- leftmost index bound of array REGISTER

OUTPUT'FANOUT           -- number of signals driven by port OUTPUT

CLK'DELAYED(5ns)        -- signal CLK delayed by 5ns
```

CHAPTER 7

EXPRESSIONS

The rules applicable to the different forms of expression, and to their evaluation, are given in this chapter.

7.1 Expressions

An expression is a formula that defines the computation of a value.

```
expression ::=
      relation { and relation }
    | relation { or relation }
    | relation { xor relation }
    | relation [ nand relation ]
    | relation [ nor relation ]

relation ::=
    simple_expression [ relational_operator simple_expression ]

simple_expression ::=
    [ sign ] term { adding_operator term }

term ::=
    factor { multiplying_operator factor }

factor ::=
    primary [ ** primary ]
    | abs primary
    | not 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, the value of the primary is the value of the object.

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 Section 10.5). For each predefined operator, the operand and result types are given in the following section.

Note:

The syntax for an expression involving logical operators allows a sequence of **and**, **or**, or **xor** operators, since the corresponding operations are associative. For operators **nand** and **nor**, however, such a sequence is not allowed, since the corresponding operations are not associative.

7.2 Operators

The operators that may be used in expressions are defined below. 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.

```
logical_operator        ::=  and | or | nand | nor | xor

relational_operator     ::=  =  | /=  | <  | <=  | >  | >=

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. For a sequence of operators with the same precedence level, the operators are associated with their operands in textual order, from left to right. The precedence of an operator is fixed and may not 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 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.

7.2.1 Logical Operators

The logical operators **and**, **or**, **nand**, **nor**, **xor**, and **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. In the latter case, for the binary operators **and**, **or**, **nand**, **nor**, and **xor**, the

operands must 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. For the unary operator **not**, 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	**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.

Note:

All of the binary logical operators belong to the class of operators with the lowest precedence. The unary logical operator **not** belongs to the class of operators with the highest precedence.

7.2.2 Relational Operators

Relational operators include tests for equality, inequality and ordering of operands. The operands of each relational operator must be of the same type. The result type of each relational operator is the predefined type BOOLEAN.

Operator	Operation	Operand Type	Result Type
=	equality	any type	BOOLEAN
/=	inequality	any type	BOOLEAN
< <= > >=	ordering	any scalar type or discrete array type	BOOLEAN

The equality and inequality operators (= and /=) are defined for all types other than file types. The equality operator returns the value TRUE if the two operands are equal, and the value FALSE otherwise. The inequality operator returns the value FALSE if the two operands are equal, and 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 multi-dimensional array values, matching elements are those whose indices match in successive positions.

The 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:
  - The leftmost element of the left operand is less than that of the right; or
  - 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.

7.2.3 Adding Operators

The adding operators + and - are predefined for any numeric type and have their conventional 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 array type	same array type	same array type
		any array type	the element type	same array type
		the element type	any array type	same array type
		the element type	the element type	any array type

For concatenation, there are three cases:

1. If both operands are one-dimensional arrays, the result of the concatenation is a one-dimensional array 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). The left bound of this result is the left bound of the left operand, unless the left operand is a null array, in which case the result of the concatenation is the right operand. The direction of the result is the direction of the left operand, unless the left operand is a null array, in which case the direction of the result is that of the right operand.

2. If only one of the operands is a one-dimensional array, the result of the concatenation is given by the rules in case (1), using in place of the other operand an implicit array having this operand as its only element. The left bound of this implicit array is the left bound of the index subtype of the array, and its direction is ascending (descending) if the index subtype is ascending (descending).

3. If neither operand is a one-dimensional array, the type of the result must be known from the context. This type must be such that each operand is an element of an implicit array, and the type of this implicit array is the same as the result type. The subtype of this implicit array is determined as in case (2), and the result of the concatenation is determined as in case (1).

Signs + and - are predefined for any numeric type and have their conventional meaning: they represent the identity and negation functions respectively. For each of these unary operators, the operand and the result have the same type.

Note:

Because of the relative precedence of signs + and - in the grammar for expressions, a signed operand must 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
```

7.2.4 Multiplying Operators

The operators * and / are predefined for any integer and any floating point type and have their conventional 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 must satisfy the relation:

A = B*N + (A **mod** B)

In addition to the above, the operators * and / 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*

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)

7.2.5 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.

7.3 Operands

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 Section 6.1; the other kinds of operands are defined in the following sections.

7.3.1 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 Chapter 13, Lexical Elements; physical literals are defined in Section 3.1.3.

Enumeration literals are literals of enumeration types. They include both identifiers and character literals. Enumeration literals are defined in Section 3.1.1.

String and bit string literals are representations of one-dimensional arrays of characters. The type of a string or bit string literal must 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 must be a one-dimensional array of a character type (for string literals) or of type BIT (for bit string literals). The lexical structure of string and bit string literals are both defined in Chapter 13, Lexical Elements.

For bit string literals, and for string literals that represent non-null array values, the direction and bounds 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 Sections 13.6 and 13.7) of the string or bit string literal. For string literals that represent null array values, the direction and leftmost bound of the array value are determined as for other string literals. If the direction is ascending, then the rightmost bound is the predecessor (as given by the PRED attribute) of the leftmost bound; otherwise the rightmost bound is the successor (as given by the SUCC attribute) of the leftmost bound.

The character literals corresponding to the graphic characters contained within a string literal or a bit string literal must 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 literal of type BIT_STRING
"54LS281"              -- a literal of type STRING
" "                    -- a string literal representing a null array
```

7.3.2 Aggregates

An aggregate 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. 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 no associations may follow an **others** association). Aggregates containing a single element association must 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. An element association with the single choice **others** is allowed in either: it specifies all remaining elements, if any.

Each element of the value defined by an aggregate must be represented once and only once in the aggregate.

The type of an aggregate must 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 must be a composite type. The type of an aggregate in turn determines the required type for each of its elements.

7.3.2.1 Record Aggregates

If the type of an aggregate is a record type, the element names given as choices must denote elements of that record type. If the choice **others** is given as a choice of a record aggregate, it must 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 must have the type of the associated record elements.

7.3.2.2 Array Aggregates

For an aggregate of a one-dimensional array type, each choice must specify values of the index type, and the expression of each element association must be of the element type. 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. A string or bit string literal is allowed in a multi-dimensional aggregate at the place 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 must 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 subtype of an array aggregate that has an **others** choice must be determinable from the context. That is, an array aggregate with an **others** choice may only appear:

1. As an actual associated with a formal parameter or formal generic declared to be of a constrained array subtype;

2. As the default expression defining the default initial value of a port declared to be of a constrained array subtype;

3. As the result expression of a function, where the corresponding function result type is a constrained array subtype;

4. As a value expression in an assignment statement, where the target is a declared object, and the subtype of the target is a constrained array subtype;

5. As the expression defining the initial value of a constant or variable object, where that object is declared to be of a constrained array subtype;

6. As the expression defining the initial value of the drivers of one or more signals in an initialization specification, where the corresponding subtype is a constrained array subtype;

7. As the expression defining the value of an attribute in an attribute specification, where that attribute is declared to be of a constrained array subtype;

8. As the operand of a qualified expression whose type mark denotes a constrained array subtype;

9. As a subaggregate of a multi-dimensional aggregate, where that aggregate itself appears in one of these contexts.

The bounds of an array that does not have an **others** choice are determined as follows. If the aggregate appears in one of the above contexts, then the direction of the index subtype of the aggregate is that of the corresponding constrained array subtype; otherwise, the direction of the index subtype of the aggregate is that of the index subtype of the base type of the aggregate. For an aggregate that has named associations, the leftmost and rightmost bounds are determined by the direction of the index subtype 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 above contexts; 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 subtype and the number of elements.

7.3.3 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 must specify exactly one corresponding actual parameter. This actual parameter is specified either explicitly, by an association element in the association list, or in the absence of such an association element, by a default expression (see Section 4.3.3).

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 must belong to the subtype of the associated formal parameter. (If the formal parameter is of an unconstrained array type, then the formal parameter takes on the subtype of the actual parameter.) The function body is executed using the actual parameter values and default expression values as the values of the corresponding formal parameters.

7.3.4 Qualified Expressions

A qualified expression 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 must 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 checks that its value belongs 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.

7.3.5 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. The type of the operand of a type conversion must be determinable independent of the context (in particular, 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.

Explicit type conversions are allowed between closely related types. In particular, conversion of an operand of a given type to the type itself is allowed. The other allowed explicit type conversions are as follows:

a. Abstract Numeric Types

The operand can be of any integer or floating point type. The value of the operand is converted to the target type, which must 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.

b. Array Types

The conversion is allowed if the operand type and the target type are array types that satisfy the following conditions:

- the types have the same dimensionality;
- for each index position, the index types are either the same or are convertible to each other; and
- the element types are the same.

If the type mark denotes an unconstrained array type, then, for each index position, the bounds of the result are obtained by converting the bounds of the operand to the corresponding index type of the target type. If the type mark denotes a constrained array subtype, then the bounds of the result are those imposed by the type mark. In either case, the value of each element of the result is that of the matching element of the operand (see Section 7.2.2).

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 the case of conversions between array types, a check is made that any constraint on the element subtype is the same for the operand array type as for the target array type. If the type mark denotes an unconstrained array type, 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 type mark denotes a constrained array subtype, 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 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 real 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:

Two array types may be closely related even if corresponding index positions have different directions.

7.3.6 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 must 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 index constraint. If an allocator includes a subtype indication and if the type of the object created is an array type, then the subtype indication must either denote a constrained subtype or include an explicit index constraint. A subtype indication that is part of an allocator must not include a resolution function.

If the type of the created object is an array type, then the created object is always 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 must 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:

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.

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'(15ns, null)                       -- initial value is specified
new NODE'(Delay => 5ns, 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
```

7.4 Static Expressions

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*, because its value is dependent only upon declarations that are local to the containing design unit, or packages used by that design unit. 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*, because its value may be dependent upon declarations that appear in other design units within the hierarchy, or upon the process of elaboration itself. A locally static expression is also considered to be globally static.

An expression is said to be locally static if and only if every operator in the expression denotes a predefined operator whose operands and result are scalar and every primary in the expression is a *locally static primary*, where a locally static primary is defined to be one of the following:

1. a literal of any type;

2. a constant (other than a deferred constant) explicitly declared by a constant declaration with a locally static subtype and initialized with a locally static expression;

3. a function call whose function name denotes a predefined operator, and whose actual parameters are each locally static expressions;

4. a predefined attribute of a locally static subtype that is a value;

5. a predefined attribute of a locally static subtype that is a function, where the actual parameter is a locally static expression;

6. a user-defined attribute whose value is defined by a locally static expression;

7. a qualified expression whose type mark denotes a locally static subtype and whose operand is a locally static expression;

8. a locally static expression enclosed in parentheses.

A locally static range is a range whose bounds are locally static expressions. 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 subtype is a constrained array subtype formed by imposing on an unconstrained array type a locally static index constraint.

A locally static subtype is either a locally static scalar subtype or a locally static array subtype.

An expression is said to be globally static if and only if every operator in the expression denotes a predefined operator and every primary in the expression is a *globally static primary*, where a globally static primary is defined to be one of the following:

1. a locally static primary;

2. a generic constant;

3. a generate parameter;

4. a constant (including a deferred constant) explicitly declared by a constant declaration with a globally static subtype and initialized with a static expression;

5. an aggregate of a globally static subtype whose element associations contain only globally static expressions;

6. a function call whose function name denotes a predefined operator, and whose actual parameters are each globally static expressions;

7. a predefined attribute of a globally static subtype that is a value or a range;

8. a predefined attribute of a globally static subtype that is a function, where the actual parameter is a globally static expression;

9. a user-defined attribute whose value is defined by a globally static expression;

10. a qualified expression whose type mark denotes a globally static subtype and whose operand is a globally static expression;

11. a globally static expression enclosed in parentheses.

A globally static range is a range whose bounds are globally static expressions. 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 subtype is a constrained array subtype formed by imposing on an unconstrained array type a globally static index constraint.

A globally static subtype is either a globally static scalar subtype or a globally static array subtype.

Note:

An expression that is required to be a static expression may 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, or an array subtype that is required to be static may either be locally static or globally static.

7.5 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 that of the most accurate predefined floating point type supported by the implementation, apart from *universal_real* itself. Furthermore, if a universal expression is a static expression, then the evaluation must be exact.

For the evaluation of an operation of a non-static universal expression, the following rules apply. If the result is of type *universal_integer*, then the values of the operands and the result must 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 must lie within the range of the floating point type with the widest range provided by the implementation, excluding type *universal_real* itself.

CHAPTER 8

SEQUENTIAL STATEMENTS

The various forms of sequential statement are described in this chapter. 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
    | signal_assignment_statement
    | variable_assignment_statement
    | procedure_call_statement
    | if_statement
    | case_statement
    | loop_statement
    | next_statement
    | exit_statement
    | return_statement
    | null_statement
```

Certain 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.

8.1 Wait Statement

The wait statement causes the suspension of a process statement or a procedure.

```
wait_statement ::=
    wait [sensitivity_clause] [condition_clause] [timeout_clause] ;

sensitivity_clause ::= on sensitivity_list

sensitivity_list ::= signal_name { , signal_name }

condition_clause ::= until condition

condition ::= boolean_expression

timeout_clause ::= for time_expression
```

The sensitivity clause defines the *sensitivity set* of the wait statement, i.e., 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 must be a static signal name, and each name must denote a signal for which reading is permitted. If no sensitivity clause appears, the sensitivity set will contain the signals denoted by the longest static prefix of each signal name that appears as a primary in the condition of the condition clause.

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 must 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 Section 2.2) 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 may also resume 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 TRUE, the process will resume. If the value of the condition is FALSE, the process will re-suspend. Such re-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.

8.2 Assertion Statement

An assertion statement checks that a specified condition is true and reports an error if it is not.

```
assertion_statement ::=
    assert condition
        [ report expression ]
        [ severity expression ] ;
```

If the **report** clause is present, it must include an expression of predefined type STRING that specifies a message to be reported. If the **severity** clause is present, it must 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 default value for the message string is "Assertion violation.". 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.

Evaluation 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, if present, of the corresponding assertion 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:

1. an indication that this message is from an assertion;
2. the value of the severity level;
3. the value of the message string;
4. the name of the design unit (see Section 11.1) containing the assertion.

8.3 Signal Assignment Statement

A signal assignment statement modifies the projected output waveforms contained in the drivers of one or more signals (see Section 9.2.1).

```
signal_assignment_statement ::=
    target <= [ transport ] waveform ;

target ::=
    name
    | aggregate

waveform ::=
    waveform_element { , waveform_element }
```

If the target of the signal assignment statement is a name, then the name must denote a signal, and the base type of the value component of each transaction produced by a waveform element on the right-hand side must be the same as the base type of the signal denoted by that name. This form of signal assignment assigns right-hand side values to the drivers associated with a single (scalar or composite) signal.

If the target of the signal assignment statement is in the form of an aggregate, then the type of the aggregate must be determinable from the context, excluding the aggregate itself, but including the fact that the type of the aggregate must be a composite type. The base type of the value component of each transaction produced by a waveform element on the right-hand side must be the same as the base type of the aggregate. Furthermore, the expression in each element association of the aggregate must be a locally static name that denotes a signal. This form of signal assignment assigns subelements of the right-hand side values to the drivers associated with the signal named as the corresponding subelement of the aggregate.

If the target of a signal assignment statement is in the form of an aggregate, and 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 a choice that is a discrete range.

The right-hand side of a signal assignment may optionally begin with the reserved word **transport**. This 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 the reserved word is not 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.

8.3.1 Updating a Projected Output Waveform

The effect of execution of a signal assignment statement is defined in terms of its effect upon the projected output waveforms (see Section 9.2.1) 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 signal 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 signal assignment statement containing a null waveform element is not a guarded signal.

The base type of the time expression in each waveform element must 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** 0ns" 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 taken together with 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 signal 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.

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 signal 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:

1. 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;

2. The new transactions are then appended to the projected output waveform in the order of their projected occurrence.

If the reserved word **transport** does not appear in the corresponding signal assignment statement, then the initial delay is considered to be inertial delay, and the projected output waveform is further modified as follows:

1 All of the new transactions are marked;

2. An old transaction is marked if it immediately precedes a marked transaction and its value component is the same as that of the marked transaction;

3. The transaction that determines the current value of the driver is marked;

4. 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 signal 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.

If a given procedure is declared by a declarative item that is not contained within a process statement, and a signal assignment statement appears in that procedure, then the target of the assignment statement must be a formal parameter of the given procedure or of a parent of that procedure, or an aggregate of such formal parameters.

Note:

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.

The above rules guarantee that the driver affected by a signal assignment statement is always statically determinable if the signal 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 signal assignment must appear within a procedure, and the affected driver is the one passed to the procedure along with a signal parameter of that procedure.

8.4 Variable Assignment Statement

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 must be of the same type.

```
variable_assignment_statement ::=
    target := expression ;
```

If the target of the variable assignment statement is a name, then the name must denote a variable, and the base type of the expression on the right-hand side must be the same as the base type of the variable denoted by that name. 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 must be determinable from the context, excluding the aggregate itself, but including the fact that the type of the aggregate must be a composite type. The base type of the expression on the right-hand side must be the same as the base type of the aggregate. Furthermore, the expression in each element association of the aggregate must be a locally static name that denotes a variable. This form of variable assignment assigns each subelement of the right-hand side value to the variable named as the corresponding subelement of the aggregate.

If the target of a variable assignment statement is in the form of an aggregate, and 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, 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.

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 an array (in which case the assignment involves a subtype conversion). Finally, the value of the expression becomes the new value of the variable.

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 of the right-hand side value that corresponds to one of the names in the aggregate. The value of the subelement of the right-hand side value then becomes the new value of the variable denoted by the corresponding name.

An error occurs if the above-mentioned subtype checks fail.

The determination of the type of the target of a variable assignment statement may require determination of the type of the expression if the target is a name that can be interpreted as the name of a variable designated by the access value returned by a function call, and similarly, as an element or slice of such a variable.

Note:

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.

Assignment to a variable of a file type is not allowed.

8.4.1 Array Variable Assignments

If the target of an assignment statement is a name denoting an array variable (including a slice), the value assigned to the target is implicitly converted to the subtype of the array variable; the result of this subtype conversion becomes the new value of the array variable.

This means that the new value of each element of the array variable is specified by the matching element (see Section 7.2.2) in the corresponding array value obtained by evaluation of the expression. The subtype conversion checks that for each element of the array variable there is a matching element in the array value, and vice versa. An error occurs if this check fails.

Note:

The implicit subtype conversion described above for assignment to an array variable is performed only for the value of the right-hand side expression as a whole; it is not performed for subelements that are array values.

8.5 Procedure Call Statement

A procedure call invokes the execution of a procedure body.

```
procedure_call_statement ::=
    procedure_name [ ( actual_parameter_part ) ] ;
```

The procedure name specifies the procedure body to be invoked. 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 must specify exactly one corresponding actual parameter. This actual parameter is specified either explicitly, by an association element in the association list, or in the absence of such an association element, by a default expression (see Section 4.3.3).

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 must belong to the subtype of the associated formal parameter. (If the formal parameter is of an unconstrained array type, then the formal parameter takes on the subtype of the actual

parameter.) The procedure body is executed using the actual parameter values and default expression values as the values of the corresponding formal parameters.

8.6 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 condition then
        sequence_of_statements
    { elsif condition then
        sequence_of_statements }
    [ else
        sequence_of_statements ]
    end if;
```

An expression specifying a condition must be of type BOOLEAN.

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.

8.7 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 expression is
        case_statement_alternative
        { case_statement_alternative }
    end case;

case_statement_alternative ::=
    when choices =>
        sequence_of_statements
```

The expression must be of a discrete type, or of a one-dimensional character array type (whose values are representable as string or bit string literals). This type must be determinable independently of the context in which the expression occurs, but using the fact that the expression must be of a discrete type or a one-dimensional character array type. Each choice in a case statement alternative must 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 locally static, whether a scalar type or an array type, then each value of the subtype must 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.

If the expression is of a one-dimensional character array type, then the expression must be the name of an object whose subtype is locally static, or it must be a qualified expression or type conversion whose type mark denotes a locally static subtype. In such a case, each choice appearing in any of the case statement alternatives must be a locally static expression whose value is of the same length as that of the case expression.

For other forms of expression, each value of the (base) type of the expression must 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 statement must 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 is not allowed as a choice of a case statement alternative.

The execution of a case statement consists of the evaluation of the expression followed by the execution of the chosen sequence of statements.

Note:

The execution of a case statement chooses one and only one alternative, since the choices are exhaustive and mutually exclusive. Qualification of the expression of a case statement by a locally static subtype can often be used to limit the number of choices that need be given explicitly.

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*.

Overloading the operator "=" has no effect on the semantics of case statement execution.

8.8 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 must repeat the label at the beginning of the loop statement.

A loop statement without an iteration scheme specifies repeated execution of the sequence of statements. Execution of the loop statement is complete when the loop is left as a consequence of the execution of a next statement, an exit statement, or a return statement.

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 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 must 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 execution of the loop statement is 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). 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.

8.9 Next Statement

A next statement is used to complete the execution of one of the iterations of an enclosing loop statement (called "loop" in what follows). The completion is conditional if the statement includes a condition.

```
next_statement ::=
    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.

8.10 Exit Statement

An exit statement is used to complete the execution of an enclosing loop statement (called "loop" in what follows). The completion is conditional if the statement includes a condition.

```
exit_statement ::=
    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.

8.11 Return Statement

A return statement is used to complete the execution of the innermost enclosing function or procedure body.

```
return_statement ::=
    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 must not have an expression. A return statement appearing in a function body must have an expression.

The value of the expression defines the result returned by the function. The type of this expression must 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 means other than the execution of a return statement.

For the execution of a return statement, the expression (if any) is first evaluated and a check is made that the value belongs to the result subtype. The execution of the return statement is thereby completed if the check succeeds; so also is the execution of the enclosing subprogram. An error occurs at the place of the return statement if the check 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.

8.12 Null Statement

A null statement performs no action.

```
null_statement ::= 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 explicitly specify that no action is to be performed when certain conditions are true. This is particularly useful in conjunction with the case statement, in which all possible values of the case expression must be covered by choices: for certain choices, it may be that no action is required.

CHAPTER 9

CONCURRENT STATEMENTS

The various forms of concurrent statement are described in this chapter. 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
    | concurrent_assertion_statement
    | concurrent_signal_assignment_statement
    | component_instantiation_statement
    | generate_statement
```

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, or block statement.

9.1 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_expression ) ]
            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 expression 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 expression defines the value of that signal at any given time (see Section 12.6.3). The type of the guard expression must be type BOOLEAN. Signal GUARD may be used to control the operation of certain statements within the block (see Section 9.5).

The implicit signal GUARD must 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 Section 1.1.1.1 and 1.1.1.2); the generic map and port map aspects define the association of actuals with those formals (see Section 5.2.1.2). Such actuals are evaluated in the context of the enclosing declarative region.

If a label appears at the end of a block statement, it must repeat the block label.

Note:

The value of signal GUARD is always defined within the scope of a given block, and 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.

An actual appearing in a port association list of a given block can never denote a formal port of the same block.

9.2 Process Statement

A process statement defines an independent sequential process representing the behavior of some portion of the design.

```
process_statement ::=
    [ process_label : ]
        process [ ( sensitivity_list ) ]
            process_declarative_part
        begin
            process_statement_part
        end process [ process_label ] ;

process_declarative_part ::=
    { process_declarative_item }

process_declarative_item ::=

    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause

process_statement_part ::=
    { sequential_statement }
```

If a 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 of the wait statement is that following the reserved word **process**. Such a process statement may not contain an explicit wait statement. Similarly, if such a process statement is a parent of a procedure, then that procedure may not contain a wait statement.

Only static signal names (see Section 6.1) may appear in the sensitivity list of a process statement.

If a label appears at the end of a process statement, it must repeat the process label.

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. Such a process, or any concurrent statement equivalent to such a process, may appear in the entity statement part of an entity declaration.

Note:

The above rules 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.

9.2.1 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 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 is defined by the default value associated with the signal (see Section 4.3.1.2).

For any driver, there is exactly one transaction whose time component is not greater than the current simulation time. The *current value* of the driver is the value component of this transaction. If, as the result of the advance of time, the current time becomes equal to the time component of the next transaction, then the first transaction is deleted from the projected output waveform, and the next becomes the current value of the driver.

9.3 Concurrent Procedure Call

A concurrent procedure call represents a process containing the corresponding sequential procedure call.

```
concurrent_procedure_call ::=
    [ label : ] procedure_call_statement
```

For any concurrent procedure call, there is an equivalent process statement. The equivalent process statement 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. Each formal parameter of a procedure that is invoked by a concurrent procedure call must be of class constant or signal.

If there exists a primary that denotes a signal in the actual part of any association element in the concurrent procedure call, and that signal 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 contains the longest static prefix of each signal name appearing as a primary in an actual part and associated with such a formal parameter; 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 procedure call is equivalent to execution of the equivalent process statement.

Example:

```
CheckTiming (tPLH, tPHL, Clk, D, Q);          -- a concurrent procedure call

process     -- the equivalent process
begin
    CheckTiming (tPLH, tPHL, Clk, D, Q);
    wait on Clk, D, Q;
end process;
```

Note:

Concurrent procedure calls make it possible to declare procedures representing commonly used processes and to easily create such processes 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 may 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.

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.

9.4 Concurrent Assertion Statement

A concurrent assertion statement represents a passive process statement containing the specified assertion statement.

```
concurrent_assertion_statement ::=
    [ label : ] assertion_statement
```

For any concurrent assertion statement, there is an equivalent process statement. The equivalent process statement 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 primary 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 contains the longest static prefix of each signal name appearing as a primary in 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.

Note:

Since a concurrent assertion statement represents a passive process statement, such a process has no outputs, and 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.

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.

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.

9.5 Concurrent Signal Assignment Statement

A concurrent signal assignment statement represents an equivalent process statement that assigns values to signals.

```
concurrent_signal_assignment_statement ::=
      [ label : ] conditional_signal_assignment
    | [ label : ] selected_signal_assignment

options ::= [ guarded ] [ transport ]
```

There are two forms of the concurrent signal assignment statement. For each form, the characteristics that distinguish it are discussed below.

Each form may include one or both of the two options **guarded** and **transport**. The option **guarded** 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 its inputs. (The signal GUARD may be one of the implicitly declared GUARD signals associated with block statements that have guard expressions, or it may be an explicitly declared signal of type Boolean that is visible at the point of the concurrent signal assignment

statement.) The option **transport** specifies that the signal assignment statement has transport delay.

If the target of a concurrent signal assignment is a name that denotes a guarded signal (see Section 4.3.1.2), 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:

1. If a label appears on the concurrent signal assignment statement, then the same label appears on the process statement.

2. If the option **transport** appears in the conditional signal assignment, then the reserved word **transport** appears in every signal assignment statement in the process statement; otherwise, it appears in no signal assignment statement in the process statement.

3. The statement part of the equivalent process statement contains a *signal transform*, which is either a sequential signal assignment statement, or an if or case statement containing sequential signal assignment statements, one for each of the alternative waveforms. The signal transform determines which of the alternative waveforms is to be assigned to the output signals. In addition, the statement part may contain a sequence of *disconnection statements*, which assign null transactions to the target of the concurrent signal assignment under certain conditions.

   If the option **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 the target of the concurrent signal assignment is a guarded target, then the statement part of the equivalent process statement 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 the target of the concurrent signal assignment is *not* a guarded target, then the statement part of the equivalent process statement is as follows:

   ```
   if GUARD then
       signal_transform
   end if;
   ```

Finally, if the concurrent signal assignment statement is *not* a guarded assignment, and the target of the concurrent signal assignment is *not* a guarded target, then the statement part of the equivalent process statement 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.

4. If the concurrent signal assignment statement is a guarded assignment, or if there exists a primary that denotes a signal in any expression (other than time expressions) within the concurrent signal assignment statement, then the process statement contains a final wait statement with an explicit sensitivity clause. The sensitivity clause contains the longest static prefix of each signal name (if any) appearing as a primary in one of the above-mentioned 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.

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 Section 5.3).

If the target of a concurrent signal assignment statement is in the form of an aggregate, then the same transformation applies. Such a target may only contain locally static signal names, and no two signal names may identify the same object, or subelement thereof.

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:

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.

9.5.1 Conditional Signal Assignment

The conditional signal assignment represents a process statement in which the signal transform is an if statement.

```
conditional_signal_assignment ::=
    target <= options conditional_waveforms ;

conditional_waveforms ::=
    { waveform when condition else }
    waveform
```

For a given conditional signal assignment, there is an equivalent process statement corresponding to it as defined for any concurrent signal assignment statement. If the conditional signal assignment is of the form

```
target <= options      waveform1 when condition1 else
                       waveform2 when condition2 else
                              .
                              .
                              .
                       waveformN-1 when conditionN-1 else
                       waveformN ;
```

then the signal transform in the corresponding process statement is of the form

```
if condition1 then
    target <= [ transport ] waveform1 ;
elsif condition2 then
    target <= [ transport ] waveform2 ;
        .
        .
        .
elsif conditionN-1 then
    target <= [ transport ] waveformN-1 ;
else
    target <= [ transport ] waveformN ;
end if ;
```

If the conditional waveform is only a single waveform, the signal transform in the corresponding process statement is of the form

```
target <= [ transport ] waveform ;
```

The characteristics of the waveforms and conditions in the conditional assignment statement must be such that the if statement in the equivalent process statement is a legal statement.

9.5.2 Selected Signal Assignment

The selected signal assignment represents a process statement in which the signal transform is a case statement.

```
selected_signal_assignment ::=
    with expression select
        target <= options selected_waveforms ;
```

```
selected_waveforms ::=
    { waveform when choices , }
    waveform when choices
```

For a given selected signal assignment, there is an equivalent process statement corresponding to it as defined for any concurrent signal assignment statement. If the selected signal assignment is of the form

```
with expression select
    target <= options   waveform1          when choice_list1 ,
                        waveform2          when choice_list2 ,
                            .
                            .
                            .
                        waveformN-1        when choice_listN-1,
                        waveformN          when choice_listN ;
```

then the signal transform in the corresponding process statement is of the form

```
case expression is
    when choice_list1 =>
        target <= [ transport ] waveform1 ;

    when choice_list2 =>
        target <= [ transport ] waveform2 ;
            .
            .
            .
    when choice_listN-1 =>
        target <= [ transport ] waveformN-1 ;

    when choice_listN =>
        target <= [ transport ] waveformN ;
end case ;
```

The characteristics of the select expression, the waveforms and the choices in the selected assignment statement must be such that the case statement in the equivalent process statement is a legal statement.

9.6 Component Instantiation Statement

A component instantiation statement defines a subcomponent of the design entity in which it appears and associates signals with the ports of that subcomponent. This subcomponent is one instance of a class of components defined by a corresponding component declaration.

```
component_instantiation_statement ::=
    instantiation_label :
        component_name
            [ generic_map_aspect ]
            [ port_map_aspect ] ;
```

The component name must be the name of a component declared in a component declaration. The generic map aspect, if present, associates a single actual with each local generic (or subelement thereof) in the corresponding component declaration. Each local generic (or subelement thereof) must be associated exactly once. Similarly, the port map aspect, if present, associates a single actual with each local port (or subelement thereof) in the corresponding component declaration. Each local port (or subelement thereof) must be associated exactly once. The generic map and port map aspects are described in Section 5.2.1.2.

Note:

A configuration specification (see Section 5.2) 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 entity.

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.

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. 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.

9.6.1 Instantiation of a Component

A component instantiation statement and a corresponding configuration specification, 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.

A component instantiation statement 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, respectively. 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 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, respectively.

For example, consider the following component declaration, instantiation, and corresponding configuration specification:

```
component COMP port (A,B : inout BIT);

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 := 1ms;
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 C is declared 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 := 1ms;        -- entity declarative item
              signal P3: Bit;                     -- arch. declarative item
          begin
              CheckTiming (P1, P2, 2*Delay);      -- entity statement
              P3 <= P1 after Delay;               -- arch. 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 Chapter 12).

9.7 Generate Statement

A generate statement provides a mechanism for iterative or conditional elaboration of a portion of a description.

```
generate_statement ::=
    generate_label :
        generation_scheme generate
            { concurrent_statement }
        end generate [ generate_label ] ;

generation_scheme ::=
    for generate_parameter_specification
    | if condition

label ::= identifier
```

If a label appears at the end of a generate statement, it must repeat the generate label.

For a generate statement with a **for** generation scheme, 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 elaboration of a generate statement is described in Section 12.4.2.

Example:

```
B: 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 B;
```

CHAPTER 10

SCOPE AND VISIBILITY

The rules defining the scope of declarations and the rules defining which identifiers are visible at various points in the text of the description are presented in this chapter. The formulation of these rules uses the notion of a declarative region.

10.1 Declarative Region

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:

1. An entity declaration, together with a corresponding architecture body.
2. A configuration declaration.
3. A subprogram declaration, together with the corresponding subprogram body.
4. A package declaration, together with the corresponding body (if any).
5. A record type declaration.
6. A component declaration.
7. A block statement.
8. A process statement.
9. A loop statement.
10. A block configuration.
11. A component configuration.

In each of the above 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).

10.2 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 entity declared by the declaration. Furthermore, if the declaration associates some notation (either an identifier, a character literal, or an operator symbol) with the declared entity, this portion of the text is also called the scope of this notation. Within the scope of an entity, and only there, there are places where it is legal to use the associated notation in order to refer to the declared entity. These places are defined by the rules of visibility and overloading.

The scope of a declaration that occurs immediately within a declarative region extends from the beginning of the declaration to the end of the declarative region; this part of the scope of a declaration is called the *immediate scope*. Furthermore, for any of the declarations listed below, the scope of the declaration extends beyond the immediate scope:

1. A declaration that occurs immediately within a package declaration.

2. An element declaration in a record type declaration.

3. A formal parameter declaration in a subprogram declaration.

4. A local generic declaration in a component declaration.

5. A local port declaration in a component declaration.

6. A formal generic declaration in an entity declaration.

7. A formal port declaration in an entity declaration.

In the absence of a separate subprogram declaration, the subprogram specification given in the subprogram body acts as the declaration and rule (3) applies also in such a case. In each of these cases, 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 above rules, the scope of *any* declaration that 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) 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 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.

Note:

The above scope rules apply to all forms of declaration. In particular, they apply also to implicit declarations.

10.3 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 chapter include any identifier other than a reserved word. The places considered in this chapter are those where a lexical element (such as an identifier) occurs. The overloaded declarations considered in this chapter 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. Two cases may arise in determining the meaning of such a declaration:

1. 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.

2. 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.

A declaration is only visible within a certain part of its scope; this part starts at the end of the declaration except in the declaration of a design unit, in which case it starts immediately after the reserved word **is** given after the identifier of the design unit.

Visibility is either by selection or direct. A declaration is visible *by selection* at places that are defined as follows:

1. For a primary unit contained in a library: at the place of the suffix in a selected name whose prefix denotes the library.

2. 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.

3. For a declaration given in a package declaration: at the place of the suffix in a selected name whose prefix denotes the package.

4. 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.

5. For a predefined attribute that applies to a given range of definition: at the place of the attribute designator (after the delimiter ') in an attribute name whose prefix belongs to the given range of definition.

6. For a user-defined attribute: at the place of the attribute designator (after the delimiter ') in an attribute name whose prefix denotes an entity with which that attribute has been associated.

7. For a formal parameter declaration of a given subprogram declaration: at the place of the formal designator (before the compound delimiter =>) in a named parameter association list of a corresponding subprogram call.

8. For a local generic declaration of a given component declaration: at the place of the formal designator (before the compound delimiter =>) in a named generic association list of a corresponding component instantiation statement; similarly, at the place of the actual designator (after the compound delimiter =>) in a generic association list of a corresponding binding indication.

9. For a local port declaration of a given component declaration: at the place of the formal designator (before the compound delimiter =>) in a named port association list of a corresponding component instantiation statement; similarly, at the place of the actual designator (after the compound delimiter =>) in a port association list of a corresponding binding indication.

10. For a formal generic declaration of a given entity declaration: at the place of the formal designator (before the compound delimiter =>) in a named generic association list of a corresponding binding indication.

11. For a formal port declaration of a given entity declaration: at the place of the formal designator (before the compound delimiter =>) in a named port association list of a corresponding binding specification.

Finally, within the declarative region associated with a construct other than a record type declaration, any declaration that occurs immediately within the region is visible by selection at the place of the suffix of an expanded name whose prefix denotes the construct.

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 below. In addition, a declaration occurring immediately within the visible part of a package can be made directly visible by means of a use clause according to the rules described in Section 10.4.

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 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.

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 must not be homographs, unless exactly one of them is the implicit declaration of a predefined operation. In such cases, a predefined operation 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; the implicit declaration is visible neither by selection nor directly.

Whenever a declaration with a certain identifier is visible from a given point, the identifier and the declared 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.

Example:

```
L1: block
        signal A,B: Bit ;
    begin
    L2: block
            signal B: Bit ;              -- an inner homograph of B
        begin
            A <= B after 5ns;            -- means L1.A <= L2.B
            B <= L1.B after 10ns;        -- means L2.B <= L1.B
        end block ;
        B <= A after 15ns;               -- means L1.B <= L1.A
    end block ;
```

10.4 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 or operator symbol, then the selected name identifies only the declaration(s) of that simple name or operator symbol contained within the package or library denoted by the prefix of the selected name. 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.

For each use clause, 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 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 declarative region associated with the design unit. The scope of a use clause may additionally extend into a configuration declaration (see Section 10.2).

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 two cases:

1. A potentially visible declaration is not made directly visible if the place considered is within the immediate scope of a homograph of the declaration.

2. Potentially visible declarations that have the same designator are not made directly visible unless each of them is either an enumeration literal specification or the declaration of a subprogram (either by a subprogram declaration or by an implicit declaration).

Note:

The above rules guarantee that a declaration that is made directly visible by a use clause cannot hide an otherwise directly visible declaration.

If an 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 entities that are actually declared in package Q will be potentially visible in design unit R (in the absence of any other use clauses).

10.5 The Context of Overload Resolution

Overloading is defined for subprograms and for enumeration literals.

For overloaded entities, overload resolution determines the actual meaning that an occurrence of an enumeration 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.

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; a *complete context* is either a declaration, a specification, or a statement.

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 described below.

1. Any rule that requires a name or expression to have a certain type, or to have the same type as another name or expression.

2. 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, real, physical, universal, character, or boolean type.

3. Any rule that requires a prefix to be appropriate for a certain type.

4. The rules that require the type of an aggregate to be determinable solely from the enclosing complete context. Similarly, the rules that require the type of the prefix of

an attribute, the type of the expression of a case statement, or the type of the operand of a type conversion to be determinable independently of the context.

5. The rules given for the resolution of overloaded subprogram calls, for the implicit conversions of universal expression; for the interpretation of discrete ranges with bounds having a universal type; and for the interpretation of an expanded name whose prefix denotes a subprogram.

CHAPTER 11

DESIGN UNITS AND THEIR ANALYSIS

The overall organization of descriptions, as well as their analysis and subsequent definition in a design library, are discussed in this chapter.

11.1 Design Units

Certain constructs may be 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

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.

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 must have a simple name that is unique within the given library, and each architecture body associated with a given entity declaration must 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 Chapter 1, Design Entities and Configurations. Package declarations and package bodies are discussed in Chapter 2, Subprograms and Packages.

11.2 Design Libraries

A *design library* is an implementation-dependent storage facility for previously analyzed design units. A given implementation may have 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 at the beginning of a design unit. There is a certain region of text called the *scope* of a library clause; this region starts immediately after the library clause, and it extends to the end of the declarative region associated with the design unit in which the library clause appears. Within this scope, except where hidden, 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 libraries in the host environment may or may not be the same. A given implementation must 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 may be 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 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 package STANDARD and package TEXTIO reside; these are the only standard packages defined by the language (see Chapter 14). (The use clause makes all declarations within package STANDARD directly visible within the corresponding design unit; see Section 10.4). Library logical name WORK denotes the current working library during a given analysis.

A secondary unit corresponding to a given primary unit may only 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).

It is recommended that library STD contain only those library units that correspond to design units defined as part of the Language Reference Manual. This set of units may change over time as the language evolves; however, portability of designs will be enhanced if, for any given version of the language, library STD contains a known set of library units.

11.3 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
```

A library clause defines library logical names that may be referenced in the design unit; library clauses are described in Section 11.2. A use clause makes certain declarations directly visible within the design unit; use clauses are described in Section 10.4.

Dependencies among design units are defined by use clauses; that is, a design unit that explicitly or implicitly mentions other library units in a use clause *depends* on those library units. These dependencies affect the allowed order of analysis of design units, as explained in Section 11.4.

Note:

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.

11.4 Order of Analysis

The rules defining the order in which design units can be analyzed are direct consequences of the visibility rules. In particular:

1. A primary unit whose name is referenced within a given design unit must be analyzed prior to the analysis of the given design unit.

2. A primary unit must be analyzed prior to the analysis of any corresponding secondary unit.

The order in which design units are analyzed must be consistent with the partial ordering defined by the above 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 must be reanalyzed before they can be used again.

CHAPTER 12

ELABORATION AND EXECUTION

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 to ultimately elaborate declarative items that are declared within those constructs.

In order to execute a model, the design hierarchy defining the model must first be elaborated. Initialization of nets 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.

12.1 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.

A design hierarchy may be defined by a design entity. Elaboration of a design hierarchy defined in this manner consists of the elaboration of the block statement equivalent to the external block defined by the design entity. The equivalent block statement is defined in Section 9.6.1. Elaboration of a block statement is defined in Section 12.4.1.

A design hierarchy may also be defined by a configuration. 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.

Elaboration of a block statement involves first elaborating each not yet elaborated package used within the block. (A package is used within a given construct if the name of that package appears within the construct, either standing alone or as the prefix of an expanded name.) Similarly, elaboration of a given package involves first elaborating each not yet elaborated package used within the given package. Elaboration of a package additionally consists of the elaboration of the declarative part of the package declaration, followed by elaboration of the declarative part of the corresponding package body, if any. Elaboration of a declarative part is defined in Section 12.3.

12.2 Elaboration of a Block Header

Elaboration of a block header consists of the elaboration of the generic clause, the generic map clause, the port clause, and the port map clause, in that order.

12.2.1 The 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 consists of elaborating the subtype indication and then creating a generic constant of that subtype.

The value of a generic constant is not defined until a subsequent generic map clause is evaluated, or in the absence of a generic map clause, until the default expression associated with the generic constant is evaluated to determine the value of the constant.

12.2.2 The Generic Map Clause

Elaboration of a generic map clause 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; the actual part of such an implicit association element is the default expression appearing in the declaration of that generic constant.

Elaboration of a generic association list consists of the elaboration of each generic association element in the association list. Elaboration of a generic association element consists of the elaboration of the formal part and the evaluation of the actual part. The generic constant or subelement thereof designated by the formal part is then initialized with the value resulting from the evaluation of the corresponding actual part.

12.2.3 The 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 consists of elaborating the subtype indication and then creating a port of that subtype.

12.2.4 The Port Map Clause

Elaboration of a port map clause consists of elaborating the port association list.

Elaboration of a port association list consists of the elaboration of each port association element in the association list. Elaboration of a port association element consists of the elaboration of the formal part; the port or subelement thereof designated by the formal part is then associated with the signal designated by the actual part. This association involves a check that the restrictions on port associations (see Section 1.1.1.2) 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 no association element associates a signal with that port, then the default expression is evaluated and the value of the port is set to the value of the expression.

12.3 Elaboration of a Declarative Part

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.

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 must be defined at the time the expression is evaluated.

Note:

It is a consequence of the above rule that the name of a signal declared within a block cannot be referenced in expressions appearing in declarative items within a block, 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.

12.3.1 Elaboration of a Declaration

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 the declaration that declares the item. 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.

12.3.1.1 Subprogram Declarations and Bodies

Elaboration of a subprogram declaration involves the elaboration of the parameter interface list of the subprogram declaration; this 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 a subprogram body has no effect other than to establish that the body can from then on be used for the execution of calls of the subprogram.

12.3.1.2 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 array type declaration, however, elaboration consists of the elaboration of the equivalent anonymous unconstrained array 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 subtype indication. 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 unconstrained array type definition 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.

12.3.1.3 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:

1. The constraint is first elaborated.

2. A check is then made that the constraint is compatible with the type or subtype denoted by the type mark (see Sections 3.1 and 3.2.1.1).

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 a size constraint consists of the evaluation of the expression.

12.3.1.4 Object Declarations

Elaboration of an object declaration that declares an object other than a file object proceeds as follows:

1. The subtype indication is first elaborated. This establishes the subtype of the object.

2. If the object declaration includes an explicit initialization expression, then the initial value of the object is obtained by evaluating the expression. Otherwise, any implicit initial value for the object is determined.

3. The object is created.

4. 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 an array 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 array type.

The elaboration of a file object declaration consists of the elaboration of the subtype indication followed by the creation of the object. The file logical name is then evaluated, and the corresponding host file is associated with the file object.

Note:

These rules apply to all object declarations other than port and generic declarations, which are elaborated as outlined in Sections 12.2.1 through 12.2.4.

The expression initializing a constant object need not be a static expression.

12.3.1.5 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 object. The creation of an alias for an array 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.

12.3.1.6 Attribute Declarations

Elaboration of an attribute declaration has no effect other than to create a template for defining attributes of items.

12.3.1.7 Component Declarations

Elaboration of a component declaration has no effect other than to create a template for instantiating component instances.

12.3.2 Elaboration of a Specification

Elaboration of a specification has the effect of associating additional information with a previously declared item.

12.3.2.1 Attribute Specifications

Elaboration of an attribute specification proceeds as follows:

1. The entity specification is elaborated in order to determine which items are affected by the attribute specification.

2. The expression is evaluated to determine the value of the attribute.

3. A new instance of the designated attribute is created and associated with each of the affected items.

4. 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 constrained array type, an implicit subtype conversion is first applied as for an assignment statement. No such conversion is necessary for an attribute of an unconstrained array 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.

12.3.2.2 Configuration Specifications

Elaboration of a configuration specification proceeds as follows:

1. The component specification is elaborated in order to determine which component instances are affected by the configuration specification.

2. The binding indication is elaborated to identify the design entity to which the affected component instances will be bound.

3. 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.

12.3.2.3 Disconnection Specifications

Elaboration of a disconnection specification proceeds as follows:

1. The guarded signal specification is elaborated in order to identify the signals affected by the disconnection specification.

2. The time expression is evaluated to determine the disconnection time for drivers of the affected signals.

3. 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.

12.4 Elaboration of a Statement Part

Concurrent statements appearing in the statement part of a block must 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.

12.4.1 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 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 consists of each of the configuration specifications implied by component configurations (see Section 1.3.2) occurring immediately within the block configuration, and in the order in which the component configurations themselves appear.

12.4.2 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 statement part consists of the 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 generate statement with a for generation scheme, 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:

1. The label of the block statement is the same as the label of the generate statement.

2. The block declarative part contains 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.

3. The block statement part consists of a copy of the concurrent statements contained within the generate statement.

For a generate statement with an if generation scheme, elaboration consists of the evaluation of the boolean expression, followed by the generation of exactly one block statement if the expression evaluates to TRUE, and no block statement otherwise. If generated, the block statement has the following form:

1. The block label is the same as the label of the generate statement.

2. The block declarative part is empty.

3. The block statement part consists of a copy of the concurrent statements contained within the generate statement.

Note:

The repetition of the block labels in the case of a for generation scheme 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.

12.4.3 Component Instantiation Statements

Elaboration of a component instantiation statement has no effect unless the component instance is either fully bound to a design entity defined by an entity declaration and architecture body or it is 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 Section 9.6.1.

12.4.4 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:

1. The process declarative part is elaborated.
2. The drivers required by the process statement are created.
3. The initial transaction defined by the default value associated with each scalar signal driven by the process statement is inserted into the corresponding driver.

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.

12.5 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:

1. 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 Section 8.8). This elaboration creates the loop parameter and evaluates the discrete range.

2. Execution of a subprogram call involves the elaboration of the parameter interface list of the corresponding subprogram declaration; this involves the elaboration of each interface declaration to create the corresponding formal parameters. Actual parameters are then associated with formal parameters. Finally, the declarative part of the corresponding subprogram body is elaborated, and the sequence of statements in the subprogram body is executed.

3. 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:

It is a consequence of the above 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.

12.6 Execution of a Model

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.

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(T)) 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 a variable representing the 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 that variable 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 expression 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.

12.6.1 Propagation of Signal Values

As simulation time advances, the transactions in the projected output waveform of a given driver (see Section 9.2.1) will each, in succession, become the value of the driver. When a driver acquires a new value in this way, regardless of whether the new value is different from the previous value, that driver is said to be *active* during that simulation cycle. A signal is said to be *active* during a given simulation cycle:

- if one of its sources is active;
- if one of its subelements is active;
- if the signal is named in the formal part of an association element in a port association list, and the corresponding actual is active;
- if the signal is a subelement of a resolved signal, and the resolved signal is active.

If a signal of a given composite type has a source that is of a different type (and therefore a type conversion function 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 type conversion function 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 above, 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 any given simulation cycle. 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 type conversion functions are involved in the propagation of signal values.

For a scalar signal S, the driving value of S is determined as follows:

- If S has no source, then the driving value of S is given by the default value associated with S (see Section 4.3.1.2).
- If S has one source that is a driver, and S is not a resolved signal (see Section 4.3.1.2), then the driving value of S is the 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 Section 4.3.3.2). The driving value of a formal part is obtained by evaluating the formal part, using the driving value of the signal denoted by the formal designator in place of the formal designator.
- If S is a resolved signal, then the driving value of S is the same as the resolved value of S 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 driver of S whose current value is determined by a null transaction (see Section 8.3.1).

For a composite signal R, the driving value of R is equal to the aggregate of the driving values of each of the scalar subelements of R.

For a scalar signal S, the *effective value* of S is determined in the following manner:

- If S is a signal declared by a signal declaration, a port of mode **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 Section 4.3.3.2). 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 Section 4.3.1.2).

For a composite signal R, the effective value of R is the aggregate of the effective values of each of the subelements of R.

For a scalar signal S, both the driving and effective values must 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 must be a matching element in both the driving and the resolved 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 current value of the signal with the newly determined effective value, as follows:

1. If S is a signal of some type that is not an array type, 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.

2. If S is an array 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.

If updating a signal causes the current value of that signal to change, then an *event* is said to have occurred on the signal. This definition applies to any updating of a signal, whether such updating occurs according to the above rules or according to the rules for updating implicit signals given in Section 12.6.2. The occurrence of an event may cause the resumption and subsequent execution of certain processes during the simulation cycle in which the event occurs.

For any signal other than one declared with the signal kind **register**, the driving and effective values of the signal are determined and the current value of that signal is updated as described above in every simulation cycle. A signal declared with the signal kind **register** is updated in the same fashion during every simulation cycle except those in which all of its sources have current values that are determined by null transactions.

Implicit signals S'Stable(T), S'Quiet(T), and S'Transaction, for any prefix S and any time T, are not updated according to the above rules; such signals are updated according to the rules described in Section 12.6.2.

Note:

In a simulation cycle, a subelement of a composite signal may be quiet, but the signal itself may be active.

The rules concerning association of actuals with formals (see Section 4.3.3.2) imply that, if a composite signal is associated with a composite port of mode **out**, **inout**, or **buffer**, and no type conversion function 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.

The algorithm for computing the driving value of a scalar signal S is recursive. For example, if S is a local signal appearing in a port association list, the driving value of S can only be obtained after the driving value of the corresponding actual part is computed. This may involve multiple executions of the above algorithm.

Similarly, 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 must 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.

No effective value is specified for **out** and **linkage** ports, since these ports may not be read.

12.6.2 Updating Implicit Signals

The kernel process updates the value of each implicit signal GUARD associated with a block statement that has a guard expression. 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 expression contains a reference to a signal S, and S is active during the current simulation cycle. In such a case, the implicit signal GUARD is updated by evaluating the corresponding guard expression and assigning the result of that evaluation to the variable representing the current value of the signal.

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.

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.

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.

The current value of a given implicit signal 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 expression 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.

These rules define a partial ordering on all signals within a model. The updating of implicit 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, 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:

The above 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, the above rules imply that, if an event occurs on S during a given simulation cycle, and 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.

12.6.3 The Simulation Cycle

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.

The initialization phase consists of the following steps:

1. The driving value and the effective value of each explicitly declared signal is computed, and the current value of the signal is set to the effective value. This value is assumed to have been the value of the signal for an infinite length of time prior to the start of simulation.

2. The value of each implicit signal of the form S'Stable(T) or S'Quiet(T) is set to True.

3. The value of each implicit GUARD signal is set to the result of evaluating the corresponding guard expression.

4. Each process in the model is executed until it suspends.

At the beginning of simulation, current time is assumed to be 0ns.

A simulation cycle consists of the following steps:

1. If no driver is active, then simulation time advances to the next time at which a driver becomes active or a process resumes. Simulation is complete when time advances to TIME'High.

2. Each active explicit signal in the model is updated. (Events may occur on signals as a result.)

3. Each implicit signal in the model is updated. (Events may occur on signals as a result.)

4. For each process P, if P is currently sensitive to a signal S, and an event has occurred on S in this simulation cycle, then P resumes.

5. Each process that has just resumed is executed until it suspends.

Note:

The initial value of any implicit signal of the form S'Transaction is not defined.

Updating of explicit signals is described in Section 12.6.1; updating of implicit signals is described in Section 12.6.2.

When a process resumes, it is added to the set of processes to be executed during the current simulation cycle. However, no process actually begins to execute until the last step of the simulation cycle, at which point all executable processes for this simulation cycle have been identified.

CHAPTER 13

LEXICAL ELEMENTS

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 chapter.

13.1 Character Set

The only characters allowed in the text of a VHDL description are the graphic characters and format effectors. Each graphic character corresponds to a unique code of the ISO seven-bit coded character set (ISO 646-1983*), and is represented (visually) by a graphical symbol. Some graphic characters are represented by different graphical symbols in alternative national representations of the ISO character set. The description of the language definition in this standard reference manual uses the ASCII graphical symbols, the ANSI graphical representation of the ISO character set.

```
graphic_character ::=
    basic_graphic_character | lower_case_letter | other_special_character

basic_graphic_character ::=
    upper_case_letter | digit | special_character | space_character

basic_character ::=
    basic_graphic_character | format_effector
```

The basic character set is sufficient for writing any description. The characters included in each of the categories of basic graphic characters are defined as follows:

(a) upper case 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

(b) digits
0 1 2 3 4 5 6 7 8 9

---

* ISO 646-1983, Information Processing--ISO 7-bit coded character set for information interchange, can be obtained from the Sales Department, American National Standardards Institute, 1430 Broadway, New York, NY 10018.

(c) special characters
" # & ' ( ) * + , - . / : ; < = > _ |

(d) the space character

Format effectors are the ISO (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:

(e) lower case 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

(f) other special characters
! $ % @ ? [\ ] ^ ` { } ~

Allowable replacements for the special characters vertical bar (|), sharp (#), and quotation (") are defined in the last section of this chapter.

Note:

The ISO character that corresponds to the sharp graphical symbol in the ASCII representation appears as a pound sterling symbol in the French, German, and United Kingdom standard national representations. In any case, the font design of graphical symbols (for example, whether they are in italic or bold typeface) is not part of the ISO standard.

The meanings of the acronyms used in this section are as follows: ANSI stands for American National Standards Institute, ASCII stands for American Standard Code for Information Interchange, and ISO stands for International Organization for Standardization.

The following names are used when referring to special characters:

symbol	name	symbol	name
"	quotation	>	greater than
#	sharp	-	underline
&	ampersand	|	vertical bar
'	apostrophe	!	exclamation mark
(	left parenthesis	$	dollar
)	right parenthesis	%	percent
*	star, multiply	?	question mark
+	plus	@	commercial at
,	comma	[	left square bracket
-	hyphen, minus	\	back-slash
.	dot, point, period	]	right square bracket
/	slash, divide	^	circumflex
:	colon	`	grave accent
;	semicolon	{	left brace
<	less than	}	right brace
=	equal	~	tilde

13.2 Lexical Elements, Separators, and Delimiters

The text of each design unit 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, or a comment.

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 any of a space character, a format effector, or the end of a line. A space character is a separator except within a comment, a string literal, or a space character literal.

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 must be format effectors other than horizontal tabulation. In any case, a sequence of one or more format effectors other than horizontal tabulation must 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. 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 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 a comment, string literal, character literal, or abstract literal.

The remaining forms of lexical elements are described in other sections of this chapter.

Note:

Each lexical element must fit on one line, since the end of a line is a separator. The quotation, sharp, and underline characters, likewise two adjacent hyphens, are not delimiters, but may form part of other lexical elements.

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, also signal assignment
<>	box

13.3 Identifiers

Identifiers are used as names and also as reserved words.

identifier ::=
letter { [underline] letter_or_digit }

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter

All characters of an identifier are significant, including any underline character inserted between a letter or digit and an adjacent letter or digit. Identifiers differing only in the use of corresponding upper and lower case letters are considered as the same.

Examples:

COUNT X c_out FFT Decoder

VHSIC X1 PageCount STORE_NEXT_ITEM

Note:

No space is allowed within an identifier since a space is a separator.

13.4 Abstract Literals

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

13.4.1 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 lower case or in upper case, with the same meaning.

An exponent indicates the power of ten 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 must 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 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.

13.4.2 Based Literals

A based literal is an abstract literal expressed in a form that specifies the base explicitly. The base must 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 ten through fifteen. A letter in a based literal (either an extended digit or the letter E of an exponent) can be written either in lower case or in upper case, with the same meaning.

The conventional meaning of based notation is assumed; in particular the value of each extended digit of a based literal must be less than the base. An exponent indicates the power of the base by which the value of the based literal with the exponent.

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

13.5 Character Literals

A character literal is formed by enclosing one of the 95 graphic characters (including the space) between two apostrophe characters. A character literal has a value that belongs to a character type.

character_literal ::= ' graphic_character '

Examples:

'A' '*' ''' ' '

13.6 String Literals

A string literal is formed by a sequence of graphic characters (possibly none) enclosed between two quotation characters 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 character itself. If a quotation character value is to be represented in the sequence of character values, then a pair of adjacent quotation characters must be written at the corresponding place within the string literal. (This means that a string literal that includes two adjacent quotation characters 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 character 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 must fit on one line since it is a lexical element (see Section 13.2). 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. Predefined type CHARACTER in package STANDARD specifies the

enumeration literals denoting both graphic and non-graphic characters. Examples of such uses of concatenation are given below:

```
"FIRST PART OF A SEQUENCE OF CHARACTERS " &
"THAT CONTINUES ON THE NEXT LINE"

"sequence that includes the" & ACK & "control character"
```

13.7 Bit String Literals

A bit string literal is formed by a sequence of extended digits enclosed between two quotations used as bit string brackets, preceded by a base specifier.

```
bit_string_literal ::=  base_specifier " bit_value "

bit_value ::=  extended_digit { [ underline ] extended_digit }

base_specifier ::=  B | O | X
```

An underline character inserted between adjacent digits of a bit string literal does not affect the value of this literal. The only letters allowed as extended digits are the letters A through F for the digits ten through fifteen. A letter in a bit string literal (either an extended digit or the base specifier) can be written either in lower case or in upper case, with the same meaning.

If the base specifier is 'B', the extended digits in the bit value are restricted to 0 and 1. If the base specifier is 'O', the extended digits in the bit value are restricted to legal digits in the octal number system, i.e., the digits 0 through 7. If the base specifier is 'X', the extended digits are all digits together with the letters A through F.

A bit string literal has a value that is a sequence of values taken from the predefined type BIT (i.e., a sequence of '0' and '1'). If the base specifier is 'B', the value of the bit string literal is the sequence given explicitly by the bit_value itself. If the base specifier is 'O' (respectively 'X'), the value of the bit string literal is the sequence obtained by replacing each extended digit in the bit_value by a sequence consisting of the three (respectively four) values representing that extended digit taken from the predefined type BIT.

The *length* of a bit string literal is the number of values of type BIT in the sequence represented.

Example:

```
X"FFF"      -- equivalent to B"1111_1111_1111"
O"777"      -- equivalent to B"111_111_111"
X"777"      -- equivalent to B"0111_0111_0111"
```

13.8 Comments

A comment starts with two adjacent hyphens and extends up to the end of the line. A comment can appear on any line of a VHDL description. 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 the enlightenment of the human reader.

Examples:

```
-- the last sentence above echoes the Algol 68 report

end; -- processing of LINE is complete

-- a long comment may be split onto
-- two or more consecutive lines

---------- the first two hyphens start the comment
```

Note:

Horizontal tabulation can be used in comments, after the double hyphen, and is equivalent to one or more spaces (see Section 13.2).

13.9 Reserved Words

The identifiers listed below are called reserved words and are reserved for significance in the language. For readability of this manual, the reserved words appear in lower case boldface.

abs	**else**	**nand**	**select**
access	**elsif**	**new**	**severity**
after	**end**	**next**	**signal**
alias	**entity**	**nor**	**subtype**
all	**exit**	**not**	
and		**null**	**then**
architecture	**file**		**to**
array	**for**	**of**	**transport**
assert	**function**	**on**	**type**
attribute		**open**	
	generate	**or**	**units**
begin	**generic**	**others**	**until**
block	**guarded**	**out**	**use**
body			
buffer	**if**	**package**	**variable**
bus	**in**	**port**	
	inout	**procedure**	**wait**
case	**is**	**process**	**when**
component			**while**
configuration	**label**	**range**	**with**
constant	**library**	**record**	
	linkage	**register**	**xor**
disconnect	**loop**	**rem**	
downto		**report**	
	map	**return**	
	mod		

A reserved word must not be used as an explicitly declared identifier.

Note:

Reserved words differing only in the use of corresponding upper and lower case letters are considered as the same (see Section 13.3). The reserved word **range** is also used as the name of a predefined attribute.

13.10 Allowable Replacements of Characters

The following replacements are allowed for the vertical bar, sharp, and quotation basic characters:

- A vertical bar character (|) can be replaced by an exclamation mark (!) where used as a delimiter.
- The sharp characters (#) of a based literal can be replaced by colons (:) provided that the replacement is done for both occurrences.
- The quotation characters (") used as string brackets at both ends of a string literal can be replaced by percent characters (%) provided that the enclosed sequence of characters contains no quotation character, and provided that both string brackets are replaced. Any percent character within the sequence of characters must then be doubled and each such doubled percent character is interpreted as a single percent character value. The same replacement is allowed for a bit string literal, provided that both bit string brackets are replaced.

These replacements do not change the meaning of the description.

Note:

It is recommended that use of the replacements for the vertical bar, sharp, and quotation characters be restricted to cases where the corresponding graphical symbols are not available. Note that the vertical bar appears as a broken bar on some equipment; replacement is not recommended in this case.

The rules given for identifiers and abstract literals are such that lower case and upper case letters can be used indifferently; these lexical elements can thus be written using only characters of the basic character set.

CHAPTER 14

PREDEFINED LANGUAGE ENVIRONMENT

This chapter describes the predefined attributes of VHDL and the packages that all VHDL implementations must provide.

14.1 Predefined Attributes

Predefined attributes denote values, functions, types, and ranges associated with various kinds of entities. These attributes are described below. For each attribute, the following information is provided:

- The kind of attribute: value, type, 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.

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'POS(X)

Kind:	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.

T'VAL(X)

Kind:	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.

T'SUCC(X)

Kind:	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'BASE'HIGH.

T'PRED(X)

Kind:	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'BASE'LOW.

T'LEFTOF(X)

Kind:	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 which is to the left of the parameter in the range of T.
Restrictions:	An error occurs if X equals T'BASE'LEFT.

T'RIGHTOF(X)

Kind:	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 which is to the right of the parameter in the range of T.
Restrictions:	An error occurs if X equals T'BASE'RIGHT.

A'LEFT [(N)]

Kind:	Function
Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype.
Parameter:	A locally static expression of type *universal_integer*, the value of which must 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 a constrained array subtype.
Parameter:	A locally static expression of type *universal_integer*, the value of which must 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 a constrained array subtype.
Parameter:	A locally static expression of type *universal_integer*, the value of which must 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 a constrained array subtype.
Parameter:	A locally static expression of type *universal_integer*, the value of which must 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 a constrained array subtype.
Parameter:	A locally static expression of type *universal_integer*, the value of which must 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 a constrained array subtype.
Parameter:	A locally static expression of type *universal_integer*, the value of which must 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:	Value
Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype.
Parameter:	A locally static expression of type *universal_integer*, the value of which must 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., the value A'HIGH(N) - A'LOW(N) + 1.

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 non-negative value. If omitted, it defaults to 0ns.
Result Type:	The base type of S.
Result:	A signal equivalent to signal S delayed T units of time. The value of S'DELAYED(t) at time Tn is always equal to the value of S at time Tn-t. Specifically:

Let R be of the same subtype as S, let T >= 0ns, 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.

(Note that S'DELAYED(0ns) is not equal to S when S has just changed.)

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 non-negative value. If omitted, it defaults to 0ns.
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 Section 12.6.2)

(Note that S'STABLE(0ns) = (S'DELAYED(0ns) = S), and S'STABLE(0ns) is FALSE only when S has just changed.)

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 non-negative value. If omitted, it defaults to 0ns.
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 Section 12.6.2)

For a given simulation cycle, S'QUIET(0ns) is TRUE if and only if S is quiet for that simulation cycle.

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.

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 scalar signal S, S'LAST_EVENT returns the largest value T of type Time for which S'DELAYED(T)'STABLE would return TRUE, if such a value for T exists; otherwise it returns 0ns.

For a composite signal S, S'LAST_EVENT returns the minimum of the values R'LAST_EVENT for every scalar subelement R of S.

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 scalar signal S, S'LAST_ACTIVE returns the largest value T of type Time for which S'DELAYED(T)'QUIET would return TRUE, if such a value for T exists; otherwise it returns 0ns.

For a composite signal S, S'LAST_ACTIVE returns the minimum of the values R'LAST_ACTIVE for every scalar subelement R of S.

S'LAST_VALUE

Kind:	Function
Prefix:	Any signal denoted by the static signal name S.
Result Type:	The base type of S.
Result:	The previous value of S, immediately before the last change of S. Specifically:

For a scalar signal S, S'LAST_VALUE = S'DELAYED(T) where T >= 0ns is the smallest value such that S'STABLE(T) is FALSE. If no such T exists, then S'LAST_VALUE is equal to S.

For a composite signal S, S'LAST_VALUE is equal to the aggregate of the previous values of each element of S.

(Note that:

(1) if S'STABLE(T) is FALSE, then by definition, for some t where 0ns <= t <= T, S'DELAYED(t) /= S; and

(2) if T_S is the smallest value such that S'STABLE (T_S) is FALSE, then for all t where 0ns <= t < T_S, S'DELAYED(t) = S.)

B'BEHAVIOR

Kind:	Value
Prefix:	Any block denoted by its block label or any design entity denoted by its architecture name.
Result Type:	Boolean.
Result:	The value is TRUE if the block defined by the block statement or by the design entity does not contain a component instantiation statement. Otherwise the value is FALSE.

B'STRUCTURE

Kind:	Value
Prefix:	Any block denoted by its block label or any design entity denoted by its architecture name.
Result Type:	Boolean.
Result:	The value is TRUE if the block defined by the block statement or by the design entity contains neither a non-passive process statement nor a concurrent statement with an equivalent process statement that is non-passive. Otherwise the value is FALSE.

Note:

The relationship between the values of the LEFT, RIGHT, LOW, and HIGH attributes is expressed in the following table:

		Ascending Range	Descending Range
T'LEFT	=	T'LOW	T'HIGH
T'RIGHT	=	T'HIGH	T'LOW

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 expressions or concurrent signal assignments. Similarly, function STANDARD.NOW should not be used in concurrent contexts.

14.2 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 may not be modified by the user.

```
package STANDARD is

    -- predefined enumeration types:

    type BOOLEAN is (FALSE, TRUE);

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

    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);
```

```
    type SEVERITY_LEVEL is (NOTE, WARNING, ERROR, FAILURE);

    -- predefined numeric types:

    type INTEGER is range implementation_defined;

    type REAL is range implementation_defined;

    -- 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;

    -- function that returns the current simulation time:

    function NOW return TIME;

    -- 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;

    type BIT_VECTOR is array (NATURAL range <>) of BIT;

end STANDARD;
```

Note:

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.

14.3 Package TEXTIO

Package TEXTIO contains declarations of types and subprograms that support formatted ASCII I/O operations.

```
package TEXTIO is

    -- Type Definitions for Text I/O

    type LINE is access STRING;              -- a LINE is a pointer to a STRING value

    type TEXT is file of STRING;             -- a file of variable-length ASCII records

    type SIDE is (RIGHT, LEFT);              -- for justifying output data within fields

    subtype WIDTH is NATURAL;                -- for specifying widths of output fields

    -- Standard Text Files

    file INPUT:  TEXT is in  "STD_INPUT";

    file OUTPUT: TEXT is out "STD_OUTPUT";

    -- Input Routines for Standard Types

    procedure READLINE (F: in TEXT; L: out 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);
```

```
    -- Output Routines for Standard Types

      procedure WRITELINE (F: out TEXT; L: in 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 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);

    -- File Position Predicates

      function ENDLINE (L: in LINE) return BOOLEAN;

    -- function ENDFILE (F: in TEXT) return BOOLEAN;

  end TEXTIO;
```

Procedures READLINE and WRITELINE 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 object designated by that value is deallocated before the new object is created. Procedure WRITELINE causes 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. If parameter L contains a null access value at the start of the call, then a null string is written to the file.

Each READ procedure declared in package TEXTIO extracts data from the beginning of the string value designated by by parameter L and modifies L so that it designates the remaining portion of the line on exit. Each WRITE procedure similarly appends data to the end of the string value designated by parameter L; in this case, however, L continues to designate the entire line being constructed. Note that write operations do not put apostrophes around single

character values or quotation marks around string values, nor do the corresponding read operations remove such additional characters if they appear in the input file.

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 gracefully recover 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 each predefined data type there is one WRITE procedure declared in package TEXTIO. Each of these has at least two parameters: L, the line to write to; and VALUE, the value to be written. Additional parameters JUSTIFIED, FIELD, DIGITS, 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. Parameter FIELD specifies the desired field width. Since the actual field width will always be at least large enough to hold the string representation of the data value, the default value 0 for the FIELD parameter has the effect of causing the data value to be written out in a field of exactly the right width (i.e., no leading or trailing spaces). Parameter JUSTIFIED specifies whether values are to be right- or left-justified within the field; the default is right-justified.

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 UNIT specifies how values of type TIME are to be formatted. The value of this parameter must 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. Thus the procedure call WRITE(Line, 5ns, UNIT=>us) would result in the string value "0.005us" being appended to the string value designated by Line, whereas WRITE(Line, 5ns) would result in the string value "5ns" being appended (since the default UNIT value is ns).

In addition to the above procedures, the predicate ENDLINE is defined for lines within a text file. For an input parameter L of type Line, function ENDLINE returns the value of the expression (L'Length = 0). 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:

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.

The execution of a read or write operation may modify or even deallocate the string object designated by input parameter L of type Line for that operation; thus 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.

APPENDIX A

SYNTAX SUMMARY

(This appendix is not a part of IEEE Std 1076-1987, IEEE Standard VHDL. It is included for information only.)

This appendix provides a summary of the syntax for VHDL. Productions are ordered alphabetically by left-hand nonterminal name. The section number indicates the section where the production is given.

abstract_literal ::= decimal_literal | based_literal **[§ 13.4]**

access_type_definition ::= **access** subtype_indication **[§ 3.3]**

actual_designator ::= **[§ 4.3.3.2]**
    expression
    | *signal*_name
    | *variable*_name
    | **open**

actual_parameter_part ::= *parameter*_association_list **[§ 7.3.3]**

actual_part ::= **[§ 4.3.3.2]**
    actual_designator
    | *function*_name (actual_designator)

adding_operator ::= + | - | & **[§ 7.2]**

aggregate ::= **[§ 7.3.2]**
    (element_association { , element_association })

alias_declaration ::= **[§ 4.3.4]**
    alias identifier : subtype_indication **is** name ;

allocator ::= **[§ 7.3.6]**
    new subtype_indication
    | **new** qualified_expression

```
architecture_body ::=                                                    [§ 1.2]
    architecture identifier of entity_name is
        architecture_declarative_part
    begin
        architecture_statement_part
    end [ architecture_simple_name ] ;

architecture_declarative_part ::=                                        [§ 1.2.1]
    { block_declarative_item }

architecture_statement_part ::=                                          [§ 1.2.2]
    { concurrent_statement }

array_type_definition ::=                                                [§ 3.2.1]
    unconstrained_array_definition  |  constrained_array_definition

assertion_statement ::=                                                  [§ 8.2]
    assert condition
        [ report expression ]
        [ severity expression ] ;

association_element ::=                                                  [§ 4.3.3.2]
    [ formal_part => ] actual_part

association_list ::=                                                     [§ 4.3.3.2]
    association_element { , association_element }

attribute_declaration ::=                                                [§ 4.4]
    attribute identifier : type_mark ;

attribute_designator ::= attribute_simple_name                           [§ 6.6]

attribute_name ::=                                                       [§ 6.6]
    prefix ' attribute_designator [ ( static_expression ) ]

attribute_specification ::=                                              [§ 5.1]
    attribute attribute_designator of entity_specification is expression ;

base ::= integer                                                         [§ 13.4.2]

base_specifier ::= B | O | X                                             [§ 13.7]

base_unit_declaration ::= identifier ;                                   [§ 3.1.3]

based_integer ::=                                                        [§ 13.4.2]
    extended_digit { [ underline ] extended_digit }

based_literal ::=                                                        [§ 13.4.2]
    base # based_integer [ . based_integer ] # [ exponent ]

basic_character ::=                                                      [§ 13.1]
    basic_graphic_character | format_effector
```

```
basic_graphic_character ::=                                          [§ 13.1]
    upper_case_letter | digit | special_character| space_character

binding_indication ::=                                               [§ 5.2.1]
    entity_aspect
    [ generic_map_aspect ]
    [ port_map_aspect ]

bit_string_literal ::= base_specifier " bit_value "                  [§ 13.7]

bit_value ::= extended_digit { [ underline ] extended_digit }        [§ 13.7]

block_configuration ::=                                              [§ 1.3.1]
    for block_specification
        { use_clause }
        { configuration_item }
    end for;

block_declarative_item ::=                                           [§ 1.2.1]
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | file_declaration
    | alias_declaration
    | component_declaration
    | attribute_declaration
    | attribute_specification
    | configuration_specification
    | disconnection_specification
    | use_clause

block_declarative_part ::=                                           [§ 9.1]
    { block_declarative_item }

block_header ::=                                                     [§ 9.1]
    [ generic_clause
    [ generic_map_aspect ; ] ]
    [ port_clause
    [ port_map_aspect ; ] ]

block_specification ::=                                              [§ 1.3.1]
    architecture_name
    | block_statement_label
    | generate_statement_label [ ( index_specification ) ]
```

```
block_statement ::=                                              [§ 9.1]
    block_label :
        block [ ( guard_expression ) ]
            block_header
            block_declarative_part
        begin
            block_statement_part
        end block [ block_label ] ;

block_statement_part ::=                                         [§ 9.1]
    { concurrent_statement }

case_statement ::=                                               [§ 8.7]
    case expression is
        case_statement_alternative
        { case_statement_alternative }
    end case ;

case_statement_alternative ::=                                   [§ 8.7]
    when choices =>
        sequence_of_statements

character_literal ::= ' graphic_character '                      [§ 13.5]

choice ::=                                                       [§ 7.3.2]
    simple_expression
    | discrete_range
    | element_simple_name
    | others

choices ::= choice { | choice }                                  [§ 7.3.2]

component_configuration ::=                                      [§ 1.3.2]
    for component_specification
        [ use binding_indication ; ]
        [ block_configuration ]
    end for ;

component_declaration ::=                                        [§ 4.5]
    component identifier
        [ local_generic_clause ]
        [ local_port_clause ]
    end component ;

component_instantiation_statement ::=                            [§ 9.6]
    instantiation_label :
        component_name
            [ generic_map_aspect ]
            [ port_map_aspect ] ;

component_specification ::=                                      [§ 5.3]
    instantiation_list : component_name
```

composite_type_definition ::= [§ 3.2]
    array_type_definition
    | record_type_definition

concurrent_assertion_statement ::= [§ 9.4]
    [label :] assertion_statement

concurrent_procedure_call ::= [§ 9.3]
    [label :] procedure_call_statement

concurrent_signal_assignment_statement ::= [§ 9.5]
      [label :] conditional_signal_assignment
    | [label :] selected_signal_assignment

concurrent_statement ::= [§ 9]
    block_statement
    | process_statement
    | concurrent_procedure_call
    | concurrent_assertion_statement
    | concurrent_signal_assignment_statement
    | component_instantiation_statement
    | generate_statement

condition ::= *boolean*_expression [§ 8.1]

condition_clause ::= **until** condition [§ 8.1]

conditional_signal_assignment ::= [§ 9.5.1]
    target <= options conditional_waveforms ;

conditional_waveforms ::= [§ 9.5.1]
    { waveform **when** condition **else** }
    waveform

configuration_declaration ::= [§ 1.3]
    configuration identifier **of** *entity*_name **is**
        configuration_declarative_part
        block_configuration
    end [*configuration*_simple_name] ;

configuration_declarative_item ::= [§ 1.3]
    use_clause
    | attribute_specification

configuration_declarative_part ::= [§ 1.3]
    { configuration_declarative_item }

configuration_item ::= [§ 1.3.1]
    block_configuration
    | component_configuration

configuration_specification ::= [§ 5.2]
    for component_specification **use** binding_indication ;

constant_declaration ::= [§ 4.3.1.1]
    constant identifier_list : subtype_indication [:= expression] ;

constrained_array_definition ::= [§ 3.2.1]
    array index_constraint **of** *element*_subtype_indication

constraint ::= [§ 4.2]
    range_constraint
    | index_constraint

context_clause ::= { context_item } [§ 11.3]

context_item ::= [§ 11.3]
    library_clause
    | use_clause

decimal_literal ::= integer [. integer] [exponent] [§ 13.4.1]

declaration ::= [§ 4]
    type_declaration
    | subtype_declaration
    | object_declaration
    | file_declaration
    | interface_declaration
    | alias_declaration
    | attribute_declaration
    | component_declaration
    | entity_declaration
    | configuration_declaration
    | subprogram_declaration
    | package_declaration

design_file ::= design_unit { design_unit } [§ 11.1]

design_unit ::= context_clause library_unit [§ 11.1]

designator ::= identifier | operator_symbol [§ 2.1]

direction ::= **to** | **downto** [§ 3.1]

disconnection_specification ::= [§ 5.3]
    disconnect guarded_signal_specification **after** *time*_expression ;

discrete_range ::= *discrete*_subtype_indication | range [§ 3.2.1]

element_association ::= [§ 7.3.2]
    [choices =>] expression

```
element_declaration ::=                                              [§ 3.2.2]
    identifier_list : element_subtype_definition ;

element_subtype_definition ::= subtype_indication                    [§ 3.2.2]

entity_aspect ::=                                                    [§ 5.2.1.1]
      entity entity_name [ ( architecture_identifier) ]
    | configuration configuration_name
    | open

entity_class ::=                                                     [§ 5.1]
    entity          | architecture    | configuration
    | procedure     | function        | package
    | type          | subtype         | constant
    | signal        | variable        | component
    | label

entity_declaration ::=                                               [§ 1.1]
    entity identifier is
        entity_header
        entity_declarative_part
 [  begin
        entity_statement_part ]
    end [ entity_simple_name ] ;

entity_declarative_item ::=                                          [§ 1.1.2]
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | file_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | disconnection_specification
    | use_clause

entity_declarative_part ::=                                          [§ 1.1.2]
    { entity_declarative_item }

entity_designator ::= simple_name | operator_symbol                  [§ 5.1]

entity_header ::=                                                    [§ 1.1.1]
    [ formal_generic_clause ]
    [ formal_port_clause ]

entity_name_list ::=                                                 [§ 5.1]
    entity_designator { , entity_designator }
    | others
    | all
```

entity_specification ::= [§ 5.1]
    entity_name_list : entity_class

entity_statement ::= [§ 1.1.3]
    concurrent_assertion_statement
    | *passive*_concurrent_procedure_call
    | *passive*_process_statement

entity_statement_part ::= [§ 1.1.3]
    { entity_statement }

enumeration_literal ::= identifier | character_literal [§ 3.1.1]

enumeration_type_definition ::= [§ 3.1.1]
    (enumeration_literal { , enumeration_literal })

exit_statement ::= [§ 8.10]
    exit [*loop*_label] [**when** condition] ;

exponent ::= E [+] integer | E - integer [§ 13.4.1]

expression ::= [§ 7.1]
      relation { **and** relation }
    | relation { **or** relation }
    | relation { **xor** relation }
    | relation [**nand** relation]
    | relation [**nor** relation]

extended_digit ::= digit | letter [§ 13.4.2]

factor ::= [§ 7.1]
    primary [** primary]
    | **abs** primary
    | **not** primary

file_declaration ::= [§ 4.3.2]
    file identifier : subtype_indication **is** [mode] file_logical_name ;

file_logical_name ::= *string*_expression [§ 4.3.2]

file_type_definition ::= [§ 3.4]
    file of type_mark

floating_type_definition := range_constraint [§ 3.1.4]

formal_designator ::= [§ 4.3.3.2]
    *generic*_name
    | *port*_name
    | *parameter*_name

formal_parameter_list ::= *parameter*_interface_list [§ 2.1.1]

formal_part ::= [§ 4.3.3.2]
    formal_designator
    | *function*_name (formal_designator)

full_type_declaration ::= [§ 4.1]
    type identifier **is** type_definition ;

function_call ::= [§ 7.3.3]
    *function*_name [(actual_parameter_part)]

generate_statement ::= [§ 9.7]
    *generate*_label :
        generation_scheme **generate**
            { concurrent_statement }
        end generate [*generate*_label] ;

generation_scheme ::= [§ 9.7]
    for *generate*_parameter_specification
    | **if** condition

generic_clause ::= [§ 1.1.1]
    generic (generic_list) ;

generic_list ::= *generic*_interface_list [§ 1.1.1.1]

generic_map_aspect ::= [§ 5.2.1.2]
    generic map (*generic*_association_list)

graphic_character ::= [§ 13.1]
    basic_graphic_character | lower_case_letter | other_special_character

guarded_signal_specification ::= [§ 5.3]
    *guarded*_signal_list : type_mark

identifier ::= [§ 13.3]
    letter { [underline] letter_or_digit }

identifier_list ::= identifier { , identifier } [§ 3.2.2]

if_statement ::= [§ 8.6]
    if condition **then**
        sequence_of_statements
    { **elsif** condition **then**
        sequence_of_statements }
    [**else**
        sequence_of_statements]
    end if ;

incomplete_type_declaration ::= **type** identifier ; [§ 3.3.1]

index_constraint ::= (discrete_range { , discrete_range }) [§ 3.2.1]

index_specification ::= [§ 1.3.1]
    discrete_range
    | *static*_expression

index_subtype_definition ::= type_mark **range** <> [§ 3.2.1]

indexed_name ::= prefix (expression { , expression }) [§ 6.4]

instantiation_list ::= [§ 5.2]
    *instantiation*_label { , *instantiation*_label }
    | **others**
    | **all**

integer ::= digit { [underline] digit } [§ 13.4.1]

integer_type_definition ::= range_constraint [§ 3.1.2]

interface_constant_declaration ::= [§ 4.3.3]
    [**constant**] identifier_list : [**in**] subtype_indication [:= *static*_expression]

interface_declaration ::= [§ 4.3.3]
    interface_constant_declaration
    | interface_signal_declaration
    | interface_variable_declaration

interface_element ::= interface_declaration [§ 4.3.3.1]

interface_list ::= [§ 4.3.3.1]
    interface_element { ; interface_element }

interface_signal_declaration ::= [§ 4.3.3]
    [**signal**] identifier_list : [mode] subtype_indication [**bus**] [:= *static*_expression]

interface_variable_declaration ::= [§ 4.3.3]
    [**variable**] identifier_list : [mode] subtype_indication [:= *static*_expression]

iteration_scheme ::= [§ 8.8]
    while condition
    | **for** *loop*_parameter_specification

label ::= identifier [§ 9.7]

letter ::= upper_case_letter | lower_case_letter [§ 13.3]

letter_or_digit ::= letter | digit [§ 13.3]

library_clause ::= **library** logical_name_list ; [§ 11.2]

library_unit ::= [§ 11.1]
    primary_unit
    | secondary_unit

literal ::= [§ 7.3.1]
    numeric_literal
    | enumeration_literal
    | string_literal
    | bit_string_literal
    | **null**

logical_name ::= identifier [§ 11.2]

logical_name_list ::= logical_name { , logical_name } [§ 11.2]

logical_operator ::= **and** | **or** | **nand** | **nor** | **xor** [§ 7.2]

loop_statement ::= [§ 8.8]
    [*loop*_label :]
        [iteration_scheme] **loop**
            sequence_of_statements
        end loop [*loop*_label] ;

miscellaneous_operator ::= ** | **abs** | **not** [§ 7.2]

mode ::= **in** | **out** | **inout** | **buffer** | **linkage** [§ 4.3.3]

multiplying_operator ::= * | / | **mod** | **rem** [§ 7.2]

name ::= [§ 6.1]
    simple_name
    | operator_symbol
    | selected_name
    | indexed_name
    | slice_name
    | attribute_name

next_statement ::= [§ 8.9]
    next [*loop*_label] [**when** condition] ;

null_statement ::= **null** ; [§ 8.12]

numeric_literal ::= [§ 7.3.1]
    abstract_literal
    | physical_literal

object_declaration ::= [§ 4.3.1]
    constant_declaration
    | signal_declaration
    | variable_declaration

operator_symbol ::= string_literal [§ 2.1]

options ::= [**guarded**] [**transport**] [§ 9.5]

```
package_body ::=                                                    [§ 2.6]
    package body package_simple_name is
        package_body_declarative_part
    end [ package_simple_name ] ;

package_body_declarative_item ::=                                   [§ 2.6]
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | file_declaration
    | alias_declaration
    | use_clause

package_body_declarative_part ::=                                   [§ 2.6]
    { package_body_declarative_item }

package_declaration ::=                                             [§ 2.5]
    package identifier is
        package_declarative_part
    end [ package_simple_name ] ;

package_declarative_item ::=                                        [§ 2.5]
    subprogram_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | file_declaration
    | alias_declaration
    | component_declaration
    | attribute_declaration
    | attribute_specification
    | disconnection_specification
    | use_clause

package_declarative_part ::=                                        [§ 2.5]
    { package_declarative_item }

parameter_specification ::=                                         [§ 8.8]
    identifier in discrete_range

physical_literal ::= [ abstract_literal ] unit_name                 [§ 3.1.3]

physical_type_definition ::=                                        [§ 3.1.3]
    range_constraint
        units
            base_unit_declaration
            { secondary_unit_declaration }
        end units
```

port_clause ::= [§ 1.1.1]
    port (port_list) ;

port_list ::= *port*_interface_list [§ 1.1.1.2]

port_map_aspect ::= [§ 5.2.1.2]
    port map (*port*_association_list)

prefix ::= [§ 6.1]
    name
    | function_call

primary ::= [§ 7.1]
    name
    | literal
    | aggregate
    | function_call
    | qualified_expression
    | type_conversion
    | allocator
    | (expression)

primary_unit ::= [§ 11.1]
    entity_declaration
    | configuration_declaration
    | package_declaration

procedure_call_statement ::= [§ 8.5]
    *procedure*_name [(actual_parameter_part)] ;

process_declarative_item ::= [§ 9.2]
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause

process_declarative_part ::= [§ 9.2]
    { process_declarative_item }

```
process_statement ::=                                                    [§ 9.2]
    [ process_label : ]
        process [ ( sensitivity_list ) ]
            process_declarative_part
        begin
            process_statement_part
        end process [ process_label ] ;

process_statement_part ::=                                               [§ 9.2]
    { sequential_statement }

qualified_expression ::=                                                 [§ 7.3.4]
    type_mark ' ( expression )
    | type_mark ' aggregate

range ::=                                                                [§ 3.1]
    range_attribute_name
    | simple_expression direction simple_expression

range_constraint ::= range range                                         [§ 3.1]

record_type_definition ::=                                               [§ 3.2.2]
    record
        element_declaration
        { element_declaration }
    end record

relation ::=                                                             [§ 7.1]
    simple_expression [ relational_operator simple_expression ]

relational_operator ::= = | /= | < | <= | > | >=                         [§ 7.2]

return_statement ::=                                                     [§ 8.11]
    return [ expression ] ;

scalar_type_definition ::=                                               [§ 3.1]
    enumeration_type_definition    | integer_type_definition
    | floating_type_definition     | physical_type_definition

secondary_unit ::=                                                       [§ 11.1]
    architecture_body
    | package_body

secondary_unit_declaration ::= identifier = physical_literal ;           [§ 3.1.3]

selected_name ::= prefix . suffix                                        [§ 6.3]

selected_signal_assignment ::=                                           [§ 9.5.2]
    with expression select
        target <= options selected_waveforms ;
```

selected_waveforms ::= [§ 9.5.2]
    { waveform **when** choices , }
    waveform **when** choices

sensitivity_clause ::= **on** sensitivity_list [§ 8.1]

sensitivity_list ::= *signal*_name { , *signal*_name } [§ 8.1]

sequence_of_statements ::= [§ 8]
    { sequential_statement }

sequential_statement ::= [§ 8]
    wait_statement
    | assertion_statement
    | signal_assignment_statement
    | variable_assignment_statement
    | procedure_call_statement
    | if_statement
    | case_statement
    | loop_statement
    | next_statement
    | exit_statement
    | return_statement
    | null_statement

sign ::= + | - [§ 7.2]

signal_assignment_statement ::= [§ 8.3]
    target <= [**transport**] waveform ;

signal_declaration ::= [§ 4.3.1.2]
    signal identifier_list : subtype_indication [signal_kind] [:= expression] ;

signal_kind ::= **register** | **bus** [§ 4.3.1.2]

signal_list ::= [§ 5.3]
    *signal*_name { , *signal*_name }
    | **others**
    | **all**

simple_expression ::= [§ 7.1]
    [sign] term { adding_operator term }

simple_name ::= identifier [§ 6.2]

slice_name ::= prefix (discrete_range) [§ 6.5]

string_literal ::= " { graphic_character } " [§ 13.6]

subprogram_body ::= [§ 2.2]
    subprogram_specification **is**
        subprogram_declarative_part
    begin
        subprogram_statement_part
    end [designator] ;

subprogram_declaration ::= [§ 2.1]
    subprogram_specification ;

subprogram_declarative_item ::= [§ 2.2]
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause

subprogram_declarative_part ::= [§ 2.2]
    { subprogram_declarative_item }

subprogram_specification ::= [§ 2.1]
    procedure designator [(formal_parameter_list)]
    | **function** designator [(formal_parameter_list)] **return** type_mark

subprogram_statement_part ::= [§ 2.2]
    { sequential_statement }

subtype_declaration ::= [§ 4.2]
    subtype identifier **is** subtype_indication ;

subtype_indication ::= [§ 4.2]
    [*resolution_function*_name] type_mark [constraint]

suffix ::= [§ 6.3]
    simple_name
    | character_literal
    | operator_symbol
    | **all**

target ::= [§ 8.3]
    name
    | aggregate

term ::= [§ 7.1]
    factor { multiplying_operator factor }

timeout_clause ::= **for** *time*_expression [§ 8.1]

type_conversion ::= type_mark (expression) [§ 7.3.5]

type_declaration ::= [§ 4.1]
    full_type_declaration
    | incomplete_type_declaration

type_definition ::= [§ 4.1]
    scalar_type_definition
    | composite_type_definition
    | access_type_definition
    | file_type_definition

type_mark ::= [§ 4.2]
    *type*_name
    | *subtype*_name

unconstrained_array_definition ::= [§ 3.2.1]
    array (index_subtype_definition { , index_subtype_definition })
        of *element*_subtype_indication

use_clause ::= [§ 10.4]
    use selected_name { , selected_name } ;

variable_assignment_statement ::= [§ 8.4]
    target := expression ;

variable_declaration ::= [§ 4.3.1.3]
    variable identifier_list : subtype_indication [:= expression] ;

wait_statement ::= [§ 8.1]
    wait [sensitivity_clause] [condition_clause] [timeout_clause] ;

waveform ::= [§ 8.3]
    waveform_element { , waveform_element }

waveform_element ::= [§ 8.3.1]
    *value*_expression [**after** *time*_expression]
    | **null** [**after** *time*_expression]

APPENDIX B

GLOSSARY

(This appendix is not a part of IEEE Std 1076-1987, IEEE Standard VHDL. It is included for information only.)

This glossary contains brief definitions for the various terms and phrases used to define the language. The complete definition of each term or phrase is provided in the main body of the LRM. For each entry, the relevant section numbers in the text is given.

Access Type. A value of an access type may designate an object created by an allocator. (§3.3)

Active Driver. A driver is said to be active during a simulation cycle in which it acquires a new value, regardless of whether the new value is different from the previous value. (§12.6.1)

Actual. An actual is either an expression, a port, a signal, or a variable associated with a formal port, formal parameter, or formal generic. (§1.1.1.2, §4.3.3.2, §5.2.1.2)

Alias. An alias is an alternate name for an object. (§4.3.4)

Aggregate. The evaluation of an aggregate yields a value of a composite type. The value is specified by giving the value of each of the elements. Either positional association or named association may be used to indicate which value is associated with which element. (§7.3.2)

Allocator. An allocator is an operation used to create anonymous, variable objects accessible by means of access values. (§3.3, §7.3.6)

Analysis. Analysis of a VHDL design file involves the syntactic and semantic analysis of source code and the insertion of intermediate form representations of design units into a design library. (§11.1)

Anonymous. Certain names are created implicitly; the simple name of such an item is not always defined, in which case the item is said to be anonymous. The base type of a numeric type or an array type is anonymous; similarly, the object denoted by an access value is anonymous. (§4.1)

Appropriate. A prefix is said to be appropriate for a type if the type of the prefix is the type considered, or the type of the prefix is an access type whose designated type is the type considered. (§6.1)

Architecture Body. An architecture body describes the internal organization or operation of a design entity. Each architecture body that is associated with a given entity declaration defines a unique design entity. An architecture may be used to describe the behavior, data flow, or structure of a design entity. (§1, §1.2)

Array Type. A value of an array type 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 indices (for a multi-dimensional array). Each index must be a value of a discrete type and must lie in the correct index range. (§3.2.1)

Ascending. A range L **to** R is called an ascending range. (§3.1)

ASCII. The American Standard Code for Information Interchange. Package Standard contains the definition of type Character, which represents the ASCII character set. (§3.1.1, §14.2)

Assertion Violation. An assertion violation occurs when the condition of an assertion statement evaluates to false. (§8.2)

Associated Driver. The associated driver for a signal assignment statement is the single driver for that signal in the (explicit or equivalent) process statement containing the signal assignment statement. (§9.2.1)

Association Element. An association element associates an actual or local with a local or formal. (§4.3.3.2)

Association List. An association list establishes correspondences between formal or local port or parameter names and local or actual names or expressions. (§4.3.3.2)

Attribute. An attribute defines some characteristic of a named entity. Some attributes are predefined and are either types, ranges, values, signals, or functions. The remaining attributes are user-defined, and are always constants. (§4.4)

Belong (to a 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. (§3.1)

Belong (to a 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. (§3)

Block. A block represents a portion of the hierarchy of a design. A block is either an external block or an internal block. (§1)

Bound. A label is said to be bound if it is identified in the instantiation list of a configuration specifications. (§5.2)

Box. 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). (§3.2.1)

Bus. A bus is one kind of guarded signal. A bus floats to a user-specified value when all of its drivers are turned off. (§4.3.3, §4.3.1.2)

Character Type. An enumeration type is said to be a character type if at least one of its enumeration literals is a character literal. (§ 3.1.1)

Compatible. 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. 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. (§3.1, §3.2.1.1).

Complete Context. A complete context is either a declaration, a specification, or a statement. (§10.5)

Composite Type. A composite type is one whose values have elements. There are two classes of composite types: array and record types. (§3, §3.2)

Concurrent Statement. Concurrent statements execute asynchronously, with no defined relative order. Concurrent statements are used for dataflow and structural description. (§9)

Configuration. A configuration describes 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. (§1, §1.3)

Conform. Two subprogram specifications, are said to 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. Conformance is similarly defined for deferred constant declarations. (§2.7)

Connected. A formal port associated with an actual port or signal is said to be connected. A formal port associated with the reserved word **open** is said to be unconnected. (§1.1.1.2)

Constant. A constant is an an object whose value may not be changed. (§4.3.1.1)

Constraint. A constraint defines a (not necessarily proper) subset of the values of a type. There are index constraints, range constraints, and size constraints. (§3)

Convertible. An operand is convertible if there exists an implicit conversion of that operation type to a given type. (§7.3.5)

Current Value. The current value of a driver is the value component of the one transaction whose time component is not greater than the current simulation time. (§9.2.1)

Declaration. A declaration defines an entity and associates an identifier (or some other notation) with the entity. This association is in effect within a region of text 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 entity; the name is said to denote the associated entity. (§4)

Default Expression. A default expression provides a default value to be used for a formal generic, 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. (§4.3.1.2, §4.3.3)

Deferred Constant. A deferred constant is a constant that is declared (by a deferred constant declaration) in a package declaration and does not have a value part; a deferred constant has a corresponding full declaration, in the corresponding package body, defining the value of the constant. (§4.3.1.1)

Denote. Where a declaration is visible, the identifier given in the declaration is said to denote the entity declared in the declaration. (§4)

Depend (on a library unit). A design unit that explicitly or implicitly mentions other library units in a use clause depends on those library units. These dependencies affect the allowed order of analysis of design units. (§11.3)

Depend (on a signal value). The current value of an implicit signal R is said to depend upon 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 expression that defines the current value of R. (§12.6.2)

Descending. A range L **downto** R is called a descending range. (§3.1)

Design Entity. An entity declaration together with an associated architecture body defines a design entity. Different design entities may share the same entity declaration, thus describing different components with the same interface, or different views of the same component. (§1)

Design File. A design file is one or more design units in sequence. (§11.1)

Design Hierarchy. A design hierarchy is a hierarchy of design entities, resulting from the successive decomposition of a design entity into subcomponents that are further decomposed. (§1)

Design Library. A design library is the host-dependent storage facility in which intermediate form representations of analyzed descriptions are stored. (§11.2)

Design Unit. A design unit may be independently analyzed and inserted into a design library. A design unit is an entity declaration, an architecture declaration, a configuration declaration, a package declaration, or a package body declaration. (§11.1)

Designate. A non-null access value is said to designate an object. (§3.3)

Designated Subtype. The designated subtype of an access type is the subtype defined by the subtype indication of the access type definition. (§3.3)

Designated Type. The designated type of an access type is the base type of the subtype defined by the subtype indication of the access type definition. (§3.3)

Discrete Array. A discrete array is a one-dimensional array whose elements are of a discrete type. (§7.2.2)

Discrete Range. A discrete range is a range whose bounds are of a discrete type. (§3.2.1.1)

Discrete Type. A discrete type is an enumeration type or an integer type. (§3.1)

Driver. A driver of a signal is a container for a projected output waveform. The signal's value 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). (§9.2.1)

Driving Value. The driving value of a signal is the value that signal provides as a source of other signals. (§12.6.1)

Effective Value. The effective value of a given signal is the value obtainable by evaluating a reference to the signal within an expression. (§12.6.1)

Elaboration. 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. (§12)

Entity Declaration. 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 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. (§1, §1.1)

Enumeration type. An enumeration type is a type whose values are defined by listing, or enumerating, them. The values are represented by enumeration literals. (§3.1, §3.1.1)

Error. An error is a condition which makes the source description illegal. If the error is detected at the time of analysis of the design unit, it prevents the creation of a library unit for the given source description. A run-time error causes simulation to terminate. (§11.4)

Erroneous. Erroneous refers to an error condition that cannot always be detected. (§2.1.1.1)

Event. An event is said to occur on a signal when the current value of the signal changes as a result of the updating of that signal with its effective value. (§12.6.1)

Execute. A process is said to execute when it performs the actions specified by the algorithm described in its statement part. (§12.6)

Expression. An expression defines the computation of a value. (§7.1)

Extend. 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). (§10.1)

External Block. An external block is a block defined by a design entity. (§1)

File Type. File types provide access to files in the host system environment. (§3, §3.4)

Floating Point Types. Floating point types approximate real numbers. (§3.1, §3.1.4)

Formal. A formal is either a formal port or formal generic of a design entity or a formal parameter of a subprogram. (§2.1, §2.1.1, §4.3.3.2, §5.2.1.2)

Full Declaration. A full constant declaration is 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. (§2.6)

Fully Bound. A component instance is fully bound if a binding indication for the component instance implies an entity and an architecture. (§5.2.1.1)

Generate Parameter. A generate parameter is declared by a generate statement. (§9.7)

Generic. A generic is a constant declared in a component declaration or an entity declaration. Unlike constants, however, the value of a generic can be supplied externally, either in a component instantiation statement or in a configuration specification. (§1.1.1.1)

Generic Interface List. A generic interface list defines local or formal generic constants. (§1.1.1.1, §4.3.3.1)

Globally Static Expression. A globally static expression is an expression which can be evaluated as soon as the design hierarchy in which it appears is elaborated. A locally static expression is also globally static. (§7.4)

Globally Static Primary. A globally static primary is a locally static primary, or one of a certain group of primaries considered to be globally static. (§7.4)

Guard. See guard expression.

Guard Expression. A guard expression is an expression associated with a block statement which controls assignment to guarded signals within the block. (§4.3.1.2, §9.1)

Guarded Assignment. A guarded assignment is a concurrent signal assignment statement that includes the option **guarded**. (§9.5)

Guarded Signal. A guarded signal is a signal declared with a signal kind (**register** or **bus**), such a signal is assigned values under the control of a Boolean-valued guard expression. (§4.3.1.2)

Guarded Signal. A guarded signal is a signal declared as a register or a bus. Such signals have special semantics when they are assigned to within guarded signal assignment statements. (§4.3.1.2)

Guarded Target. A guarded target is a signal assignment target consisting only of guarded signals. An unguarded target is a target consisting only of unguarded signals. (§9.5)

Hidden. A declaration may be hidden in its scope by a homograph of the declaration. A hidden declaration is not directly visible. (§10.3)

Homograph. 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. (§10.3)

Identify. A name in an element association in an aggregate used in an assignment target is said to identify a signal or variable and any subelements of that signal or variable. (§8.3, 8.4)

Immediate Scope. The immediate scope of a declaration that occurs immediately within a given declarative region is the portion of the scope that extends from the beginning of the declaration to the end of the declarative region. (§10.2)

Immediately Within. 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. (§10.1)

Implicit Signal. An implicit signal is any signal S'Stable(T) or S'Quiet(T), or any implicit GUARD signal. (§12.6.2)

Imply. A binding indication in a configuration specification is said to imply the design entity indicated directly, indirectly, or by default. (§5.2.1.1)

Incomplete Type Declaration. An incomplete type declaration is used to define mutually recursive access types. (§3.3.1)

Index Constraint. An index constraint determines the index range for every index of an array type, and thereby the bounds of the array. (§3.2.1.1)

Index Range. An index range is the range of values that belong to the range corresponding to an index. (§3.2.1)

Index Subtype. The index subtype for a given index position of an array is the subtype denoted by the type mark of the corresponding index subtype definition. (§3.2.1)

Inertial Delay. Inertial delay is a delay model for modeling switching circuits; a pulse whose duration is shorter than the switching time of the circuit will not be transmitted. It is the default delay mode for signal assignment statements. (§8.3)

Initial Value Expression. An initial value expression specifies the initial value to be assigned to a variable or constant. (§4.3.1.3)

Inputs. The inputs of a concurrent signal assignment statement are signals identified by the longest static prefix of each signal name appearing as a primary in each expression (other than time expressions) within the concurrent signal assignment statement. (§9.5)

Instance. A component instantiation statement represents an instance of the corresponding component. Each instance of a component may have different actuals associated with its local ports and generics. (§9.6.1)

Integer Type. The values of an integer type represent integer numbers within a specific range. (§3.1, §3.1.2)

Interface List. An interface list declares the interface objects required by a subprogram, component, design entity, or block statement. (§4.3.3.1)

Internal Block. An internal block is a nested block in a design unit, defined by a block statement. (§1)

Kernel Process. 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, detects events that occur and causes the appropriate processes to execute in response to those events. (§12.6)

To the Left Of. 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. (§3.1)

Left-to-Right Order. 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. (§3.1)

Library. See design library.

Library Unit. A library unit is the intermediate form representation of an analyzed design unit. (§11.1)

Locally Static Expression. A locally static expression is an expression which can be evaluated during the analysis of the design unit in which it appears. (§7.4)

Locally Static Name. A name is said to be locally static if every expression in the name is a locally static expression. (§6.1)

Locally Static Primary. A locally static primary is one of a certain group of primaries, which includes literals, certain constants, and certain attributes. (§7.4)

Longest Static Prefix. 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. (§6.1)

Loop Parameter. A loop parameter is declared by a loop statement. (§8.8)

Lower Bound. For a range L **to** R or L **downto** R, the smaller of L and R is called the lower bound of the range. (§3.1)

Matching Element. Matching elements are elements of two composite type values defined to correspond to each other, for certain logical and relational operations. (§7.2.2)

Mode. The mode of a port or parameter specifies the direction of information flow through the port or parameter. Modes are **in, out, inout, buffer,** or **linkage**. (§4.3.3)

Model. The elaboration of a design hierarchy produces a model that can be executed in order to simulate the design represented by the model. (§12.6)

Name. Each form of declaration associates an identifier with a declared entity. Only within its scope, there are places where it is possible to use the identifier to refer to the associated declared entity; these places are defined by the visibility rules. At such places the identifier is said to be a name of the entity. (§4, §6.1)

Named Association. An association element is said to be named if the formal designator appears explicitly. (§4.3.3.2, §7.3.2)

Null Array. If in an array's index constraint any of the discrete ranges defines a null range, then the array is a null array, having no components. (§3.2.1.1)

Null Range. A null range is a range that specifies an empty subset of values; a range L **to** R is a null range if L > R; a range L **downto** R is a null range if L < R. (§3.1)

Null Slice. A null slice is a slice whose discrete range is a null range. (§6.5)

Null Waveform Element. A null waveform element is used to turn off a driver of a guarded signal. (§8.3.1)

Null Transaction. A null transaction is a transaction produced by evaluating a null waveform element. (§8.3.1)

Numeric Type. A numeric type is either an integer type, a floating point type, or a physical type. (§3.1)

Object. An object is an entity that contains a value of a given type. There are three classes of objects: constants, signals, and variables. (§4.3)

Overloading. Identifiers or enumeration literals that denote two different entities are said to be overloaded. Enumeration literals, subprograms, and predefined operators may be overloaded. (§2.3.1, §3.1.1)

Parameter. A parameter is a constant, variable, or signal 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 data flow allowed through the parameter. (§2.1.1)

Parameter Interface List. A parameter interface list declares the parameters for a subprogram. It may contain interface constant declarations, interface signal declarations, or interface variable declarations, or any combination thereof. (§4.3.3.1)

Parameter Type Profile. 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 at each parameter position corresponding parameters have the same base type. (§2.3)

Parameter and Result Type Profile. 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 either both are functions with the same result base type, or neither of the two is a function. (§2.3)

Parent. A process or a subprogram is said to be a parent of a given procedure if that process or subprogram contains a procedure call statement for the given procedure or for a parent of the given procedure. (§2.2)

Passive Process. 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. (§9.2)

Physical Type. A physical type is used to represent measurements of some quantity. (§3.1, §3.1.3)

Port. A port is a signal declared in the interface list of an entity declaration, 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 data flow allowed through the port. (§1.1.1.2, §4.3.1.2)

Port Interface List. A port interface list declares the ports of a block, component, or design entity. It consists entirely of interface signal declarations. (§1.1.1.2, §4.3.1.1)

Positional Association. An association element is said to be positional if the formal designator does not appear explicitly; an actual designator at a given position in an association list corresponds to the interface element at the same position in the interface list. (§4.3.3.2, §7.3.2)

Primary. A primary is one of the elements making up an expression. (§7.1)

Projected Output Waveform. A projected output waveform consists of a sequence of one or more transactions, representing the current and projected future values of the driver. (§9.2.1)

Quiet. In a given simulation cycle, a signal that is not active is said to be quiet. (§12.6.1)

Range. A range specifies a subset of values of a scalar type. (§3.1)

Range Constraint. A range constraint specifies the range of values in a type. (§ 3.1, 3.1.2)

Read. The value of an object is said to be read when its value is referenced or when certain of its attributes are referenced. (§4.3.3)

Record Type. A record type is a composite type. Values of a record type consist of named elements. (§3.2.2, §7.3.2.1)

Register. A register is one kind of guarded signal. A register retains its last driven value when all of its drivers are turned off. (§4.3.1.2)

Regular Structure. A regular structure consists of 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. (§9.7)

Resolution. Resolution is the process of determining the resolved value of a resolved signal, based on the values of multiple sources for that signal. (§2.4, §4.3.1.2)

Resolution Function. A resolution function is a user-defined function that computes the resolved value of a resolved signal. (§2.4, §4.3.1.2)

Resolved Signal. A resolved signal is a signal that has an associated resolution function. (§4.3.1.2)

Resolved Value. The resolved value of a signal is the output of the resolution function associated with the resolved signal, determined as a function of the collection of inputs from the signal's multiple sources. (§2.4, §4.3.1.2)

Resource Library. A resource library is a library containing library units that are referenced within the design unit being analyzed. (§11.2)

Result Subtype. The result subtype of a function is the subtype of the returned value of the function. (§2.1)

Resume. A process that resumes is a given simulation cycle becomes ready to execute and will execute at the end of the given simulation cycle. (§12.6.3)

Satisfy. A value is said to satisfy a constraint if the value is in the subset determined by the constraint. (§3, §3.2.1.1)

Scalar Type. A scalar type is a type such that values of the type have no elements. Integer types, floating point types, physical types, and enumeration types are scalar types. (§3, §3.1)

Scope. The scope of a declaration is a portion of the text in which a declaration may be visible. This portion is defined by visibility and overloading rules. (§10.2)

Sensitivity Set. The sensitivity set of a wait statement is the set of signals to which the wait statement is sensitive. The sensitivity set is given explicitly in an **on** clause, or is implied by an **until** clause. (§8.1)

Sequential Statement. Sequential statements execute in sequence, one after another. They are used for algorithmic description. (§8)

Short-Circuit Operation. A short-circuit operation is an operation for which the right operand is evaluated if and 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. (§7.2)

Signal. A signal is 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 either signal declarations or by port declarations. (§4.3.1.2)

Simple Name. A simple name for an entity is either the identifier associated with the entity by its declaration, or another identifier associated with the entity by an alias declaration. (§6.2)

Single Object Declaration. An object declaration is called a single object declaration if its identifier list has a single identifier. (§4.3.1)

Slice. A slice is a one-dimensional array of a sequence of consecutive elements of another one-dimensional array. (§6.5)

Source. A source of a signal contributes to the signal's value. A source is either a driver or port of a component instance with which the signal is associated, or a composite collection of sources. (§4.3.1.2)

Specification. A specification associates additional information with a previously declared entity. There are four kinds of specifications: attribute specifications, initialization specifications, configuration specifications, and disconnection specifications. (§5)

Static. See locally static and globally static.

Static Name. A name is said to be a static name if every expression that appears as part of the name (for example, as an index expression) is a static expression. (§6.1)

Static Range. A static range is a range whose bounds are static expressions. (§7.4)

Static Signal Name. A static signal name is a static name that denotes a signal. (§6.1)

Subelement. The term subelement is used instead of "element" to indicate either an element or an element of another element. Where other subelements are excluded, the term "element" is used instead. (§3)

Subprogram Specification. A subprogram specification specifies the designator of the subprogram, any formal parameters of the subprogram, and the result type for a function subprogram. (§2.1)

Subtype. A subtype is a type together with a constraint. (§3)

Suspend. A process that stops executing and waits for an event or for a time period to elapse is said to be suspended. (§12.6.3)

Timeout Interval. The timeout interval of a wait statement **until** clause determines the maximum time the wait will suspend a process. (§8.1)

Transaction. A transaction is a pair consisting of a value and a time. The value part represents a (current or) future value of the driver; the time part represents the relative delay before the value becomes the current value. (§9.2.1)

Transport Delay. Transport delay is 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. (§8.3)

Type. A type is a set of values and a set of operations. (§3)

Unassociated Formal. An unassociated formal is a formal that is not associated with an actual. (§5.2.1.2)

Unconstrained Subtype. A subtype is said to be unconstrained if it corresponds to a condition that imposes no restriction. (§4.3)

Unit Name. In a physical type declaration, each unit declaration (either the base unit declaration or a secondary unit declaration) defines a unit name. (§3.1.3)

Universal_Integer. Integer literals are literals of an anonymous predefined integer type that is called *universal_integer* in this manual. (§3.1.2, §7.3.1)

Universal_Real. Floating point literals are literals of an anonymous predefined type that is called *universal_real* in this manual. (§3.1.4, §7.3.1)

Update. The value of a signal is said to be updated when the signal appears as the target of an assignment statement, (indirectly) when it is associated with an interface object of mode **out**, **buffer**, **inout**, or **linkage**, or when one of its subelements is updated. (§4.3.3)

Upper bound. For a range L **to** R or L **downto** R, the larger of L and R is called the upper bound of the range. (§3.1)

Variable. A variable is an object with a single current value. (§4.3.1.3)

Visible. A declaration of an identifier 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 an occurrence of the identifier used in the declaration. A visible declaration is visible by selection (e.g., by using an expanded name) or directly visible (e.g., by using a simple name). (§10.3)

Waveform. A waveform consists of a series of transactions. Each transaction 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. (§8.3)

Working Library. A working library is a design library into which the library unit resulting from the analysis of a design unit is placed. (§11.2)

Acknowledgements

The work of the VHDL Analysis and Standardization Group was supported by many different companies and organizations. The following list includes those that sponsored VASG meetings, those that sent representatives to VASG meetings, and those that allowed their staff the time to review and comment on preliminary drafts of this standard:

AAI Corporation
Aerospace Corporation
Air Force Institute of Technology
Air Force Wright Avionics Laboratories
AT&T Bell Labs
Boeing Aerospace
Boeing Electronics
CAE/Tektronix
Calma
Case Western University
CAD Language Systems, Inc.
Daisy Systems
Emerson Electric
ENDOT, Inc.
Gateway Design Automation
GEC Avionics, Ltd.
General Dynamics
General Electric
Harris Corporation
Hewlett-Packard
Honeywell, Inc.
Hughes Aircraft
IBM
IMAG/Grenoble
Intermetrics, Inc.
Lear Siegler, Inc.
Logic Automation
MATRA
MCC
McDonnell Douglas
Mentor Graphics
MITRE
NSA
PCA
Raytheon
Rensselaer Polytechnic University
Research Triangle Institute
Royal Systems and Radar Establishment
Silicon Solutions
Silvar-Lisco
Softool Corporation
Sperry
Stanford University
Thomson-CSF
Unisys
University of Grenoble
University of Cincinnati
University of Virginia
United Technologies Microelectronics Center
Virginia Polytechnic Institute
Valid Logic Systems
Vantage Analysis Systems
ViewLogic Systems
Warner Robins ALC
Westinghouse
Zycad