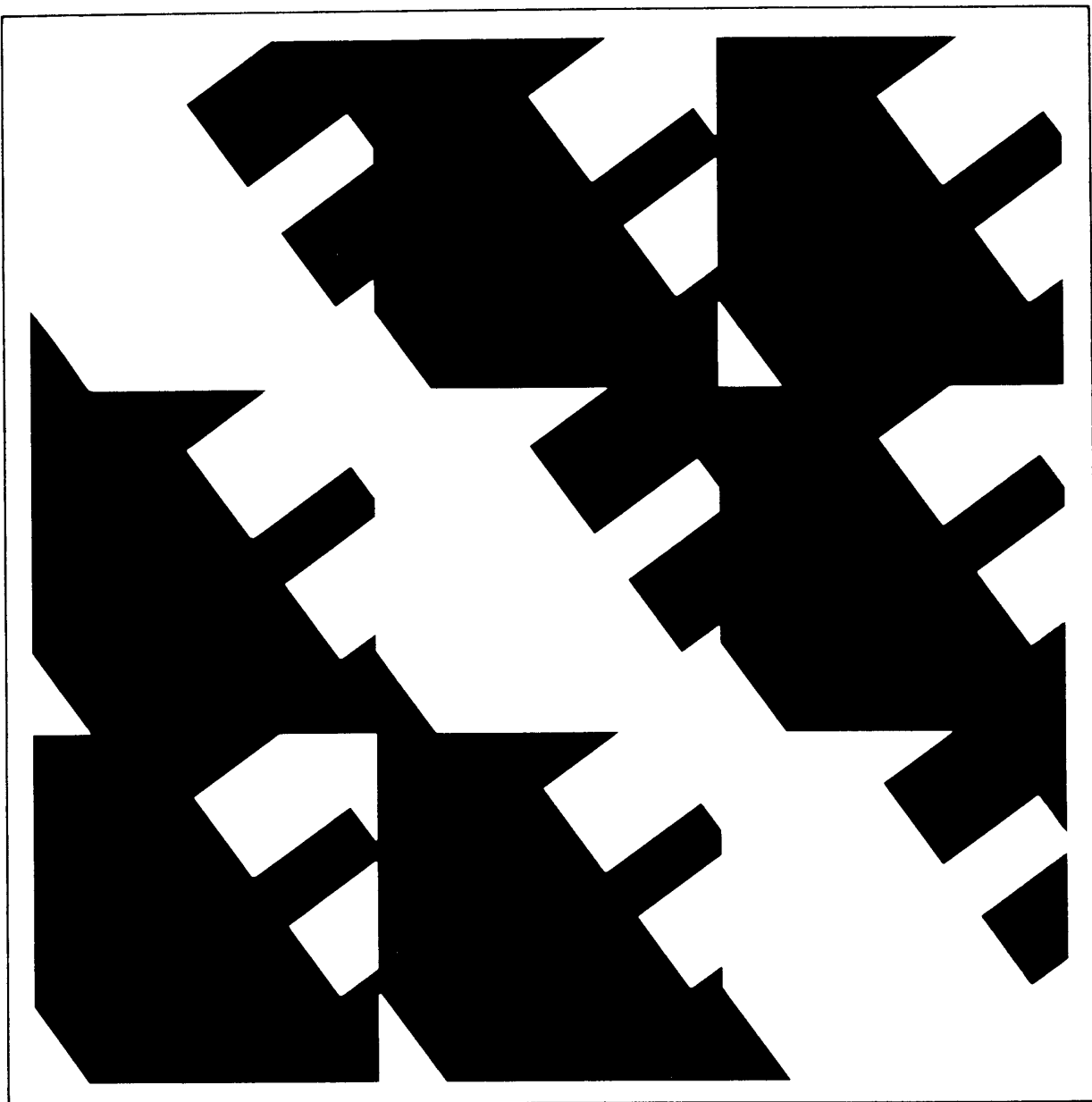


IEEE Standard VHDL Language Reference Manual

IEEE Std 1076-1987



Published by The Institute of Electrical and Electronics Engineers, Inc. 345 East 47th Street, New York, NY 10017, USA
March 21, 1988 SH11957

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**

© Copyright 1988 by

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

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

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
Youn Huh
John Jensen

Bob Johnson
Susan Johnston
George Konstantinow
Stan Krolkoski
Rick Lazansky
Jean Lester
Roger Lipsett
Shin-ming Liu
Al Lowenstein
Bruce Lundebay
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 Ada¹ 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	Don Fraser	Arthur E. Levy	William E. Russell, Jr
Guy Adam	Hingsum S. Fung	Jeff Lewis	Roy L. Russo
Gordon Adshead	Daniel Gale	John Lewis	Jere L. Sanburn
Harvey Alperin	Daniel Garvin	Charles W. Lillie	Albert Sanson
Kenneth Anderson	Sumit Ghosh	Dan Lorts	Debabrata Sarma
James R. Armstrong	Gerald Ginsberg	David B. Loveman	Paul R. Saucier
Lisa R. Asher	Michael J. Gooding	Al Lowenstein	Larry Saunders
Morris Balamut	Donald F. Gorman	Joseph F. P. Luhukay	Lowell Savage
Peter E. Barck	John Graham	Chidchanok Lursinsap	Anil D. Savkar
Daniel S. Barclay	Arnold Greenspan	Leon I. Maissel	Helmut Scheibenzuber
David Barton	T. Winston Griffin	Fred Malver	Warren M. Scheinin
Joseph C. Batz	J. Steven Grout	Erich Marschner	Steven Schlosser
Rodolfo Betancourt	Hans Grundner	Arthur J. Mason	Dick Schlotfeld
William D. Billowitch	Jeffrey Haeffele	Lawrence Maturo	Moe Shahdad
Lester I. Birman	Arthur Hann	Michael C. McFarland	Ravi Shankar
Richard J. Bonneau	Fred H. Harder	Robert J. McGowen	William H. Sherwood
Douglas B. Boyle	Michael G. Harrison	Paul J. Menchini	Isao Shirakawa
Craig A. Brown	Raymond L. Hasenstab	Jean Mermet	Sajjan G. Shiva
R. Scott Brunton	Raymond L. Heather	Alexander Miczo	Ronald J. Short
Antonius Bunsen	Robert Heatherington	Viju Monie	David M. Siefert
Raul Camposano	Kenneth E. Hermance	Kris Moorthi	James E. Sigler
Lawrence Carpenter	Jack Hilibrand	David Morris	Gabriel M. Silberman
William N. Carr	John Hines	Robert Morris	Evangelos Simoudis
Harold W. Carter	Michael Hirasuna	John W. Mowery	Alec Stanculescu
Michael T. Carter	Raymond Hookway	Lawrence J. O'Connell	Richard Summar
Ralph Cavin III	Paul W. Horstmann	Eckhard Obel	Tom H. Tabb
Edmund Cheng	Ching Hsiao	Roy Oishi	Jacques Tete
Charles R. Childress	Paul E. Hubbard	Leslie A. Orlidge	Tim L. Thorp
Shiu-Kai Chin	Youn Huh	Catherine Ozenfant	Robert Tieche
James B. Clary	John E. Jensen	Rich Palmer	William C. Topf
David Coelho	Robert Johnson	Alice C. Parker	Tuan Tran
Frank Conforti	Timothy V. Johnson	Ken Parker	Jack Trautman
Tedd Corman	William A. Johnson	Curtis Parks	Steve Trimberger
James E. Cottrell	Susan Johnston	Latit Patnaik	Joseph Tront
Scott H. Cravens	Joe Kagenski	Richard J. Patrick	Richard W. Tucker
Mr Jack L. Cross	Osamu Karatsu	Frederick D. Petry	Mark-Rene Uchida
Brian Dalio	Helmuth M. Kaunzinger	Steve Platz	Kenneth Van Bree
Michael W. Davis	Eskil Kjelkerud	Jean Pouilly	Anthony A. F. Vogelpoel
Allen M. Dewey	Berrie C. Knott	Paolo Prinetto	Rene J. Vuarnoz
Bernd Dinklage	Stanley Krolikoski	Giorgio Puggelli	L. W. Wagner
James Do	Mike Kuzemchak	Edward G. Pumphrey	Ron Waxman
Gary Duggan	Sonja V. Kval	Richard Rath	Richard J. Weger
Leo Egan, Jr.	Tuvia Lamdan	Kim Rawlinson	John Michael Williams
Ludwig D. Eggermont	Glen G. Langdon, Jr.	James R. Reeder	Ronald D. Williams
Bernd F. Eichensauer	Robert P. Larsen	Hassan K. Reghbat	Craig Winton
Abdullah C. Erdal	Lars-Olov Larsson	Johannes Reh	Chris J. Xydes
Wolf-Dieter Erdmann	Edwin Law	Daniel Rosenkrantz	Jon J. Yenger
Jim Flournoy	Rick Lazansky	Donald E. Rudisill	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* $\leq V$) and ($V \leq$ *upper bound*) are both true and the range is not a null range. The operators $>$, $<$, and $\leq$ 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 +2147483647. 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