BS IEC 61691-1-1:2011

BSI Standards Publication

Behavioural languages

Part 1-1: VHDL Language reference manual

bsi.

...making excellence a habit.™

Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

National foreword

This British Standard is the UK implementation of IEC 61691-1-1:2011. It supersedes BS IEC 61691-1-1:2004 which is withdrawn.

The UK participation in its preparation was entrusted to Technical Committee GEL/93, Design automation.

A list of organizations represented on this committee can be obtained on request to its secretary.

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for its correct application.

© BSI 2011

ISBN 978 0 580 75662 7

ICS 35.060

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 31 July 2011.

Amendments issued since publication

Amd. No.	Date	Text affected

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

IEC 61691-1-1

Edition 2.0 2011-05

INTERNATIONAL STANDARD

IEEE Std 1076™

Behavioural languages –
Part 1-1: VHDL Language Reference Manual

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

PRICE CODE XS

ICS 25.040 / 35.060

ISBN 978-2-88912-440-4

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

Contents

1. Overview of this standard 1

1.1 Scope 1
1.2 Purpose 1
1.3 Structure and terminology of this standard 2

2. Normative references 5

3. Design entities and configurations 7

3.1 General 7
3.2 Entity declarations 7
3.3 Architecture bodies 10
3.4 Configuration declarations 13

4. Subprograms and packages 19

4.1 General 19
4.2 Subprogram declarations 19
4.3 Subprogram bodies 23
4.4 Subprogram instantiation declarations 26
4.5 Subprogram overloading 26
4.6 Resolution functions 29
4.7 Package declarations 30
4.8 Package bodies 31
4.9 Package instantiation declarations 33
4.10 Conformance rules 34

5. Types 35

5.1 General 35
5.2 Scalar types 36
5.3 Composite types 44
5.4 Access types 53
5.5 File types 55
5.6 Protected types 58
5.7 String representations 61

6. Declarations 63

6.1 General 63
6.2 Type declarations 64
6.3 Subtype declarations 64
6.4 Objects 66
6.5 Interface declarations 73
6.6 Alias declarations 89
6.7 Attribute declarations 92
6.8 Component declarations 93
6.9 Group template declarations 93
6.10 Group declarations 93
6.11 PSL clock declarations 94

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

7. Specifications ... 95

7.1 General ... 95
7.2 Attribute specification ... 95
7.3 Configuration specification ... 98
7.4 Disconnection specification ... 103

8. Names ... 107

8.1 General ... 107
8.2 Simple names ... 108
8.3 Selected names ... 108
8.4 Indexed names ... 111
8.5 Slice names ... 112
8.6 Attribute names ... 112
8.7 External names ... 113

9. Expressions ... 117

9.1 General ... 117
9.2 Operators ... 118
9.3 Operands ... 131
9.4 Static expressions ... 139
9.5 Universal expressions ... 142

10. Sequential statements ... 145

10.1 General ... 145
10.2 Wait statement ... 145
10.3 Assertion statement ... 147
10.4 Report statement ... 148
10.5 Signal assignment statement ... 149
10.6 Variable assignment statement ... 160
10.7 Procedure call statement ... 163
10.8 If statement ... 164
10.9 Case statement ... 164
10.10 Loop statement ... 166
10.11 Next statement ... 167
10.12 Exit statement ... 167
10.13 Return statement ... 168
10.14 Null statement ... 168

11. Concurrent statements ... 169

11.1 General ... 169
11.2 Block statement ... 169
11.3 Process statement ... 170
11.4 Concurrent procedure call statements ... 172
11.5 Concurrent assertion statements ... 173
11.6 Concurrent signal assignment statements ... 174
11.7 Component instantiation statements ... 176
11.8 Generate statements ... 182

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

12. Scope and visibility ... 185

12.1 Declarative region ... 185
12.2 Scope of declarations ... 185
12.3 Visibility ... 187
12.4 Use clauses ... 191
12.5 The context of overload resolution ... 192

13. Design units and their analysis ... 195

13.1 Design units ... 195
13.2 Design libraries ... 195
13.3 Context declarations ... 197
13.4 Context clauses ... 197
13.5 Order of analysis ... 198

14. Elaboration and execution ... 199

14.1 General ... 199
14.2 Elaboration of a design hierarchy ... 199
14.3 Elaboration of a block, package, or subprogram header ... 202
14.4 Elaboration of a declarative part ... 205
14.5 Elaboration of a statement part ... 210
14.6 Dynamic elaboration ... 213
14.7 Execution of a model ... 214

15. Lexical elements ... 225

15.1 General ... 225
15.2 Character set ... 225
15.3 Lexical elements, separators, and delimiters ... 227
15.4 Identifiers ... 229
15.5 Abstract literals ... 230
15.6 Character literals ... 231
15.7 String literals ... 231
15.8 Bit string literals ... 232
15.9 Comments ... 234
15.10Reserved words ... 235
15.11Tool directives ... 237

16. Predefined language environment ... 239

16.1 General ... 239
16.2 Predefined attributes ... 239
16.3 Package STANDARD ... 254
16.4 Package TEXTIO ... 268
16.5 Standard environment package ... 274
16.6 Standard mathematical packages ... 275
16.7 Standard multivalue logic package ... 276
16.8 Standard synthesis packages ... 277
16.9 Standard synthesis context declarations ... 283
16.10Fixed-point package ... 283
16.11Floating-point package ... 284

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

17. VHDL Procedural Interface overview 285

17.1 General 285
17.2 Organization of the interface 285
17.3 Capability sets 286
17.4 Handles 288

18. VHPI access functions 291

18.1 General 291
18.2 Information access functions 291
18.3 Property access functions 293
18.4 Access by name function 294

19. VHPI information model 295

19.1 General 295
19.2 Formal notation 295
19.3 Class inheritance hierarchy 296
19.4 Name properties 297
19.5 The stdUninstantiated package 310
19.6 The stdHierarchy package 313
19.7 The stdTypes package 320
19.8 The stdExpr package 322
19.9 The stdSpec package 325
19.10 The stdSubprograms package 327
19.11 The stdStmts package 329
19.12 The stdConnectivity package 335
19.13 The stdCallbacks package 340
19.14 The stdEngine package 340
19.15 The stdForeign package 341
19.16 The stdMeta package 341
19.17 The stdTool package 343
19.18 Application contexts 344

20. VHPI tool execution 345

20.1 General 345
20.2 Registration phase 345
20.3 Analysis phase 351
20.4 Elaboration phase 351
20.5 Initialization phase 353
20.6 Simulation phase 353
20.7 Save phase 353
20.8 Restart phase 354
20.9 Reset phase 354
20.10 Termination phase 355

21. VHPI callbacks 357

21.1 General 357
21.2 Callback functions 357
21.3 Callback reasons 359

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

22. VHPI value access and update ... 371

22.1 General ... 371
22.2 Value structures and types ... 371
22.3 Reading object values ... 374
22.4 Formatting values ... 375
22.5 Updating object values ... 377
22.6 Scheduling transactions on drivers ... 381

23. VHPI function reference ... 385

23.1 General ... 385
23.2 vhpi_assert ... 385
23.3 vhpi_check_error ... 386
23.4 vhpi_compare_handles ... 388
23.5 vhpi_control ... 389
23.6 vhpi_create ... 390
23.7 vhpi_disable_cb ... 392
23.8 vhpi_enable_cb ... 393
23.9 vhpi_format_value ... 394
23.10vhpi_get ... 396
23.11vhpi_get_cb_info ... 396
23.12vhpi_get_data ... 397
23.13vhpi_get_foreignf_info ... 399
23.14vhpi_get_next_time ... 400
23.15vhpi_get_phys ... 401
23.16vhpi_get_real ... 402
23.17vhpi_get_str ... 402
23.18vhpi_get_time ... 403
23.19vhpi_get_value ... 404
23.20vhpi_handle ... 405
23.21vhpi_handle_by_index ... 406
23.22vhpi_handle_by_name ... 408
23.23vhpi_is_printable ... 410
23.24vhpi_iterator ... 411
23.25vhpi_printf ... 412
23.26vhpi_protected_call ... 413
23.27vhpi_put_data ... 415
23.28vhpi_put_value ... 417
23.29vhpi_register_cb ... 418
23.30vhpi_register_foreignf ... 419
23.31vhpi_release_handle ... 421
23.32vhpi_remove_cb ... 422
23.33vhpi_scan ... 422
23.34vhpi_schedule_transaction ... 423
23.35vhpi_vprintf ... 426

24. Standard tool directives ... 429

24.1 Protect tool directives ... 429

Annex A (informative) Description of accompanying files ... 447

Annex B (normative) VHPI header file ... 451

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

Annex C (informative) Syntax summary 477

Annex D (informative) Potentially nonportable constructs 501

Annex E (informative) Changes from IEEE Std 1076-2002 503

Annex F (informative) Features under consideration for removal 511

Annex G (informative) Guide to use of standard packages 513

Annex H (informative) Guide to use of protect directives 551

Annex I (informative) Glossary 557

Annex J (informative) Bibliography 585

Annex K (informative) IEEE List of participants 587

Index 589

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

INTERNATIONAL ELECTROTECHNICAL COMMISSION

BEHAVIOURAL LANGUAGES –

Part 1-1: VHDL Language Reference Manual

FOREWORD

1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.

2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.

3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.

4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.

5) IEC itself does not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC marks of conformity. IEC is not responsible for any services carried out by independent certification bodies.

6) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

International Standard IEC 61691-1-1/IEEE Std 1076 has been processed through IEC technical committee 93: *Design automation*.

This second edition cancels and replaces the first edition published in 2004. This edition constitutes a technical revision.

The text of this standard is based on the following documents:

IEEE Std	FDIS	Report on voting
1076 (2008)	93/302/FDIS	93/304/RVD

Full information on the voting for the approval of this standard can be found in the report on voting indicated in the above table.

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

A list of parts of the IEC 61691 series can be found on the IEC website.

The committee has decided that the contents of this publication will remain unchanged until the stability date indicated on the IEC web site under "http://webstore.iec.ch" in the data related to the specific publication. At this date, the publication will be

- reconfirmed,
- withdrawn,
- replaced by a revised edition, or
- amended.

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

IEC/IEEE Dual Logo International Standards

This Dual Logo International Standard is the result of an agreement between the IEC and the Institute of Electrical and Electronics Engineers, Inc. (IEEE). The original IEEE Standard was submitted to the IEC for consideration under the agreement, and the resulting IEC/IEEE Dual Logo International Standard has been published in accordance with the ISO/IEC Directives.

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

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

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

The existence of an IEC/IEEE Dual Logo International Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEC/IEEE Dual Logo International Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard.

Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

In publishing and making this document available, the IEC and IEEE are not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Neither the IEC nor IEEE is undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEC/IEEE Dual Logo International Standards or IEEE Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

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

Comments for revision of IEC/IEEE Dual Logo International Standards are welcome from any interested party, regardless of membership affiliation with the IEC or IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board, 445 Hoes Lane, Piscataway, NJ 08854, USA and/or General Secretary, IEC, 3, rue de Varembé, PO Box 131, 1211 Geneva 20, Switzerland.

Authorization to photocopy portions of any individual standard for internal or personal use is granted by the Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

IEEE Standard VHDL Language Reference Manual

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society

Approved 26 September 2008
IEEE SA-Standards Board

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

Acknowledgments

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

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

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

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

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

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

IEEE and POSIX are registered trademarks in the U.S. Patent & Trademark Office, owned by the Institute of Electrical and Electronics Engineers, Incorporated.

MagicDraw and No Magic, Inc., are registered trademarks of No Magic, Inc. in the United States and other countries.

TRI-STATE is a registered trademark of National Semiconductor Corporation.

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

IEEE introduction

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

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

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

Notice to users

Laws and regulations

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

Copyrights

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

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

Updating of IEEE documents

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

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

Errata

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

Interpretations

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

Patents

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

IMPORTANT NOTICE: This standard is not intended to assure safety, security, health, or environmental protection in all circumstances. Implementers of the standard are responsible for determining appropriate safety, security, environmental, and health practices or regulatory requirements.

This IEEE document is made available for use subject to important notices and legal disclaimers. These notices and disclaimers appear in all publications containing this document and may be found under the heading "Important Notice" or "Important Notices and Disclaimers Concerning IEEE Documents." They can also be obtained on request from IEEE or viewed at http://standards.ieee.org/IPR/disclaimers.html.

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

Behavioural languages – Part 1-1: VHDL Language Reference Manual

1. Overview of this standard

1.1 Scope

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

1.2 Purpose

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

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

1.3 Structure and terminology of this standard

1.3.1 General

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

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

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

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

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

1.3.2 Syntactic description

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

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

   formal_port_list

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

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

   array

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

c) A *production* consists of a *left-hand side*, the symbol “::=” (which is read as “can be replaced by”), and a *right-hand side*. The left-hand side of a production is always a syntactic category; the right-hand side is a replacement rule. The meaning of a production is a textual-replacement rule: any occurrence of the left-hand side may be replaced by an instance of the right-hand side.

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

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

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

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

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

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

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

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

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

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

1.3.3 Semantic description

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

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

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

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

illegal: A synonym for “error.”

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

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

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

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

1.3.5 Incorporation of Property Specification Language

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

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

2. Normative references

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

IEC 62531:2007, Standard for Property Specification Language (PSL) ¦
IEEE Std 1850™-2005, IEEE Standard for Property Specification Language (PSL)

NOTE—IEEE Std 1850-2005 was adopted as IEC 62531:2007

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

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

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

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

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

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

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

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

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

[5]ISO/IEC publications are available from the ISO Central Secretariat, Case Postale 56, 1 chemin de la Voie-Creuse, CH-1211 Genève 20, Switzerland/Suisse (http://www.iso.ch/) and from the IEC Central Office, Case Postale 131, 3 rue de Varembé, CH-1211 Genève 20, Switzerland/Suisse (http://www.iec.ch/). ISO/IEC publications are also available in the United States from Global Engineering Documents, 15 Inverness Way East, Englewood, Colorado 80112, USA (http://global.ihs.com/). Electronic copies are available in the United States from the American National Standards Institute, 25 West 43rd Street, 4th Floor, New York, NY 10036, USA (http://www.ansi.org/).

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

3. Design entities and configurations

3.1 General

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

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

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

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

3.2 Entity declarations

3.2.1 General

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

```
entity_declaration ::=
    entity identifier is
        entity_header
        entity_declarative_part
    [ begin
        entity_statement_part ]
    end [ entity ] [ entity_simple_name ] ;
```

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

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

3.2.2 Entity header

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

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

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

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

Examples:

— An entity declaration with port declarations only:

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

— An entity declaration with generic declarations also:

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

— An entity declaration with neither:

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

3.2.3 Entity declarative part

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

entity_declarative_part ::=
{ entity_declarative_item }

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

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

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

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

Example:

— An entity declaration with entity declarative items:

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

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

3.2.4 Entity statement part

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

entity_statement_part ::=
  { entity_statement }

entity_statement ::=
  concurrent_assertion_statement
| *passive*_concurrent_procedure_call_statement

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

| *passive*_process_statement
| *PSL*_PSL_Directive

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

Example:

— An entity declaration with statements:

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

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

3.3 Architecture bodies

3.3.1 General

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

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

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

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

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

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

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

3.3.2 Architecture declarative part

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

```
architecture_declarative_part ::=
    { block_declarative_item }

block_declarative_item ::=
      subprogram_declaration
    | subprogram_body
    | subprogram_instantiation_declaration
    | package_declaration
    | package_body
    | package_instantiation_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | shared_variable_declaration
    | file_declaration
    | alias_declaration
    | component_declaration
    | attribute_declaration
    | attribute_specification
    | configuration_specification
    | disconnection_specification
    | use_clause
    | group_template_declaration
    | group_declaration
    | PSL_Property_Declaration
    | PSL_Sequence_Declaration
    | PSL_Clock_Declaration
```

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

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

3.3.3 Architecture statement part

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

architecture_statement_part ::=
{ concurrent_statement }

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

Examples:

— A body of entity Full_Adder:

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

— A body of entity TestBench:

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

— A body of entity AndGate:

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

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

3.4 Configuration declarations

3.4.1 General

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

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

configuration_declarative_part ::=
  { configuration_declarative_item }

configuration_declarative_item ::=
  use_clause
  | attribute_specification
  | group_declaration

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

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

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

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

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

Examples:

— An architecture of a microprocessor:

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

— A configuration of the microprocessor:

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

3.4.2 Block configuration

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

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

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

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

configuration_item ::=
  block_configuration
 | component_configuration

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

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

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

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

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

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

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

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

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

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

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

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

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

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

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

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

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

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

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

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

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

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

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

Examples:

— A block configuration for a design entity:

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

— A block configuration for a block statement:

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

3.4.3 Component configuration

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

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

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

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

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

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

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

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

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

Examples:

— A component configuration with binding indication:

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

— A component configuration containing block configurations:

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

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

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


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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

4. Subprograms and packages

4.1 General

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

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

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

4.2 Subprogram declarations

4.2.1 General

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

subprogram_declaration ::=
    subprogram_specification ;

subprogram_specification ::=
    procedure_specification | function_specification

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

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

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

designator ::= identifier | operator_symbol

operator_symbol ::= string_literal

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

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

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

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

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

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

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

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

4.2.2 Formal parameters

4.2.2.1 Formal parameter lists

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

formal_parameter_list ::= *parameter*_interface_list

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

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

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

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

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

4.2.2.2 Constant and variable parameters

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

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

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

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

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.
Documents
Document Sharing Hub
Full document details: BS IEC 61691-1-1:2011

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

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

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

4.2.2.3 Signal parameters

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

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

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

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

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

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

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

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

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

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

4.2.2.4 File parameters

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

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

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

4.3 Subprogram bodies

A subprogram body specifies the execution of a subprogram.

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

subprogram_declarative_part ::=
    { subprogram_declarative_item }

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

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

subprogram_statement_part ::=
{ sequential_statement }

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

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

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

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

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

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

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

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

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

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

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

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

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

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

Example:

— The declaration of a foreign function subprogram:

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

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

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

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

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

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

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

4.4 Subprogram instantiation declarations

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

subprogram_instantiation_declaration ::=
    subprogram_kind designator **is new** *uninstantiated_subprogram*_name [signature]
        [generic_map_aspect] ;

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

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

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

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

4.5 Subprogram overloading

4.5.1 General

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

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

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

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

Examples:

— Declarations of overloaded subprograms:

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

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

— Calls to overloaded subprograms:

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

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

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

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

4.5.2 Operator overloading

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

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.
Documents
Document Sharing Hub
Full document details: BS IEC 61691-1-1:2011

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

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

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

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

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

Examples:

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

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

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

4.5.3 Signatures

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

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

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

— The number of type marks prior to the reserved word **return**, if any, matches the number of formal parameters of the subprogram.

— At each parameter position, the base type denoted by the type mark of the signature is the same as the base type of the corresponding formal parameter of the subprogram.

— If the reserved word **return** is present, the subprogram is a function and the base type of the type mark following the reserved word in the signature is the same as the base type of the return type of the function, or the reserved word **return** is absent and the subprogram is a procedure.

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

Example:

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

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

4.6 Resolution functions

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

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

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

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

Example:

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

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

4.7 Package declarations

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

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

package_header ::=
    [generic_clause
    [generic_map_aspect ;]]

package_declarative_part ::=
    { package_declarative_item }

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

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

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

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

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

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

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

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

Examples:

— A package declaration that needs no package body:

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

— A package declaration that needs a package body:

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

4.8 Package bodies

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

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

package_body_declarative_part ::=
  { package_body_declarative_item }

package_body_declarative_item ::=
  subprogram_declaration

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub
Full document details: BS IEC 61691-1-1:2011

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

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

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

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

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

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

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

Example:

```
package body TriState is

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

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 61691-1-1:2011

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

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

end package body TriState;
```

4.9 Package instantiation declarations

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

package_instantiation_declaration ::=
 package identifier **is new** *uninstantiated_package*_name
 [generic_map_aspect] ;

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

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

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

Published by IEC under license from IEEE. © 2008 IEEE. All rights reserved.
Documents
Document Sharing Hub
Full document details: BS IEC 61691-1-1:2011