IEC 61691-1-1

Edition 3.0 2023-10

INTERNATIONAL STANDARD

IEEE Std 1076™

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

IEC 61691-1-1:2023-10(en) IEEE Std 1076-2019

Documents
Document Sharing Hub

Full document details: IEC 61691-1-1 2023

THIS PUBLICATION IS COPYRIGHT PROTECTED
Copyright © 2019 IEEE

All rights reserved. IEEE is a registered trademark in the U.S. Patent & Trademark Office, owned by the Institute of Electrical and Electronics Engineers, Inc. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the IEC Central Office. Any questions about IEEE copyright should be addressed to the IEEE. Enquiries about obtaining additional rights to this publication and other information requests should be addressed to the IEC or your local IEC member National Committee.

IEC Secretariat
3, rue de Varembé
CH-1211 Geneva 20
Switzerland
Tel.: +41 22 919 02 11
info@iec.ch
www.iec.ch

Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue
New York, NY 10016-5997
United States of America
stds.info@ieee.org
www.ieee.org

About the IEC
The International Electrotechnical Commission (IEC) is the leading global organization that prepares and publishes International Standards for all electrical, electronic and related technologies.

About IEC publications
The technical content of IEC publications is kept under constant review by the IEC. Please make sure that you have the latest edition, a corrigendum or an amendment might have been published.

IEC publications search - webstore.iec.ch/advsearchform
The advanced search enables to find IEC publications by a variety of criteria (reference number, text, technical committee, …). It also gives information on projects, replaced and withdrawn publications.

IEC Just Published - webstore.iec.ch/justpublished
Stay up to date on all new IEC publications. Just Published details all new publications released. Available online and once a month by email.

IEC Customer Service Centre - webstore.iec.ch/csc
If you wish to give us your feedback on this publication or need further assistance, please contact the Customer Service Centre: sales@iec.ch.

IEC Products & Services Portal - products.iec.ch
Discover our powerful search engine and read freely all the publications previews. With a subscription you will always have access to up to date content tailored to your needs.

Electropedia - www.electropedia.org
The world's leading online dictionary on electrotechnology, containing more than 22 300 terminological entries in English and French, with equivalent terms in 19 additional languages. Also known as the International Electrotechnical Vocabulary (IEV) online.

Documents
Document Sharing Hub

Full document details: IEC 61691-1-1 2023

IEC 61691-1-1

Edition 3.0 2023-10

INTERNATIONAL STANDARD

IEEE Std 1076™

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

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

ICS 25.040.01, 35.060

ISBN 978-2-8322-7517-7

Warning! Make sure that you obtained this publication from an authorized distributor.

® Registered trademark of the International Electrotechnical Commission

Documents
Document Sharing Hub

Full document details: IEC 61691-1-1 2023

Contents

1. Overview ... 14

1.1 Scope ... 14
1.2 Purpose ... 14
1.3 Structure and terminology of this standard ... 14
1.4 Word usage ... 16

2. Normative references ... 18

3. Design entities and configurations ... 19

3.1 General ... 19
3.2 Entity declarations ... 19
3.3 Architecture bodies ... 22
3.4 Configuration declarations ... 25

4. Subprograms and packages ... 32

4.1 General ... 32
4.2 Subprogram declarations ... 32
4.3 Subprogram bodies ... 37
4.4 Subprogram instantiation declarations ... 39
4.5 Subprogram overloading ... 40
4.6 Resolution functions ... 43
4.7 Package declarations ... 44
4.8 Package bodies ... 45
4.9 Package instantiation declarations ... 47
4.10 Conformance rules ... 48

5. Types ... 49

5.1 General ... 49
5.2 Scalar types ... 50
5.3 Composite types ... 58
5.4 Access types ... 68
5.5 File types ... 70
5.6 Protected types ... 75
5.7 String representations ... 79
5.8 Unspecified types ... 81

6. Declarations ... 85

6.1 General ... 85
6.2 Type declarations ... 86
6.3 Subtype declarations ... 86
6.4 Objects ... 88
6.5 Interface declarations ... 96
6.6 Alias declarations ... 119
6.7 Attribute declarations ... 122
6.8 Component declarations ... 123
6.9 Group template declarations ... 123

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Full document details: IEC 61691-1-1 2023

6.10 Group declarations ... 124
6.11 PSL clock declarations ... 124

7. Specifications ... 126

7.1 General ... 126
7.2 Attribute specification ... 126
7.3 Configuration specification ... 129
7.4 Disconnection specification ... 135

8. Names ... 138

8.1 General ... 138
8.2 Simple names ... 139
8.3 Selected names ... 140
8.4 Indexed names ... 142
8.5 Slice names ... 143
8.6 Attribute names ... 143
8.7 External names ... 144

9. Expressions ... 148

9.1 General ... 148
9.2 Operators ... 149
9.3 Operands ... 163
9.4 Static expressions ... 172
9.5 Universal expressions ... 175

10. Sequential statements ... 176

10.1 General ... 176
10.2 Wait statement ... 176
10.3 Assertion statement ... 178
10.4 Report statement ... 179
10.5 Signal assignment statement ... 180
10.6 Variable assignment statement ... 189
10.7 Procedure call statement ... 191
10.8 If statement ... 192
10.9 Case statement ... 193
10.10 Loop statement ... 195
10.11 Next statement ... 196
10.12 Exit statement ... 196
10.13 Return statement ... 196
10.14 Null statement ... 197
10.15 Sequential block statement ... 197

11. Concurrent statements ... 199

11.1 General ... 199
11.2 Block statement ... 199
11.3 Process statement ... 200
11.4 Concurrent procedure call statements ... 202
11.5 Concurrent assertion statements ... 203
11.6 Concurrent signal assignment statements ... 204

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Full document details: IEC 61691-1-1 2023

11.7 Component instantiation statements ... 206
11.8 Generate statements ... 212

12. Scope and visibility ... 216

12.1 Declarative region ... 216
12.2 Scope of declarations ... 216
12.3 Visibility ... 218
12.4 Use clauses ... 222
12.5 The context of overload resolution ... 223

13. Design units and their analysis ... 225

13.1 Design units ... 225
13.2 Design libraries ... 225
13.3 Context declarations ... 227
13.4 Context clauses ... 227
13.5 Order of analysis ... 228

14. Elaboration and execution ... 229

14.1 General ... 229
14.2 Elaboration of a design hierarchy ... 229
14.3 Elaboration of a block, package, subprogram or protected type header ... 232
14.4 Elaboration of a declarative part ... 235
14.5 Elaboration of a statement part ... 241
14.6 Dynamic elaboration ... 244
14.7 Execution of a model ... 245

15. Lexical elements ... 256

15.1 General ... 256
15.2 Character set ... 256
15.3 Lexical elements, separators, and delimiters ... 258
15.4 Identifiers ... 260
15.5 Abstract literals ... 261
15.6 Character literals ... 262
15.7 String literals ... 262
15.8 Bit string literals ... 263
15.9 Comments ... 266
15.10 Reserved words ... 266
15.11 Tool directives ... 268

16. Predefined language environment ... 269

16.1 General ... 269
16.2 Predefined attributes ... 269
16.3 Package STANDARD ... 287
16.4 Package TEXTIO ... 305
16.5 Standard environment package ... 311
16.6 Standard mathematical packages ... 325
16.7 Standard multivalue logic package ... 326
16.8 Standard synthesis packages ... 326
16.9 Standard synthesis context declarations ... 332

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Full document details: IEC 61691-1-1 2023

16.10 Fixed-point package ... 333
16.11 Floating-point package ... 333
16.12 Reflection package ... 334

17. VHDL Procedural Interface overview ... 355

17.1 General ... 355
17.2 Organization of the interface ... 355
17.3 Capability sets ... 356
17.4 Handles ... 358

18. VHPI access functions ... 360

18.1 General ... 360
18.2 Information access functions ... 360
18.3 Property access functions ... 362
18.4 Access by name function ... 363

19. VHPI information model ... 364

19.1 General ... 364
19.2 Formal notation ... 364
19.3 Class inheritance hierarchy ... 365
19.4 Name properties ... 366
19.5 The stdUninstantiated package ... 379
19.6 The stdHierarchy package ... 382
19.7 The stdTypes package ... 388
19.8 The stdExpr package ... 391
19.9 The stdSpec package ... 394
19.10 The stdSubprograms package ... 396
19.11 The stdStmts package ... 398
19.12 The stdConnectivity package ... 404
19.13 The stdCallbacks package ... 409
19.14 The stdEngine package ... 409
19.15 The stdForeign package ... 410
19.16 The stdMeta package ... 410
19.17 The stdTool package ... 412
19.18 Application contexts ... 413

20. VHPI tool execution ... 414

20.1 General ... 414
20.2 Registration phase ... 414
20.3 Analysis phase ... 420
20.4 Elaboration phase ... 420
20.5 Initialization phase ... 422
20.6 Simulation phase ... 422
20.7 Save phase ... 422
20.8 Restart phase ... 423
20.9 Reset phase ... 423
20.10 Termination phase ... 424

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: IEC 61691-1-1 2023

21. VHPI callbacks ... 425

21.1 General ... 425
21.2 Callback functions ... 425
21.3 Callback reasons ... 427

22. VHPI value access and update ... 439

22.1 General ... 439
22.2 Value structures and types ... 439
22.3 Reading object values ... 442
22.4 Formatting values ... 443
22.5 Updating object values ... 445
22.6 Scheduling transactions on drivers ... 449

23. VHPI function reference ... 452

23.1 General ... 452
23.2 vhpi_assert ... 452
23.3 vhpi_check_error ... 453
23.4 vhpi_compare_handles ... 456
23.5 vhpi_control ... 457
23.6 vhpi_create ... 459
23.7 vhpi_disable_cb ... 462
23.8 vhpi_enable_cb ... 462
23.9 vhpi_format_value ... 464
23.10 vhpi_get ... 465
23.11 vhpi_get_cb_info ... 466
23.12 vhpi_get_data ... 467
23.13 vhpi_get_foreignf_info ... 469
23.14 vhpi_get_next_time ... 470
23.15 vhpi_get_phys ... 472
23.16 vhpi_get_real ... 472
23.17 vhpi_get_str ... 473
23.18 vhpi_get_time ... 474
23.19 vhpi_get_value ... 475
23.20 vhpi_handle ... 476
23.21 vhpi_handle_by_index ... 477
23.22 vhpi_handle_by_name ... 481
23.23 vhpi_is_printable ... 483
23.24 vhpi_iterator ... 484
23.25 vhpi_printf ... 485
23.26 vhpi_protected_call ... 486
23.27 vhpi_put_data ... 490
23.28 vhpi_put_value ... 492
23.29 vhpi_register_cb ... 493
23.30 vhpi_register_foreignf ... 496
23.31 vhpi_release_handle ... 499
23.32 vhpi_remove_cb ... 499
23.33 vhpi_scan ... 500
23.34 vhpi_schedule_transaction ... 501
23.35 vhpi_vprintf ... 506

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Full document details: IEC 61691-1-1 2023

24. Standard tool directives 507

24.1 Protect tool directives 507
24.2 Conditional analysis tool directives 523

Annex A (informative) VHPI definitions file 525

Annex B (normative) VHPI header file 527

Annex C (informative) Syntax summary 530

Annex D (informative) Potentially non-portable constructs 557

Annex E (informative) Changes from IEEE Std 1076-2008 558

Annex F (informative) Features under consideration for removal 559

Annex G (informative) Guide to use of standard packages 560

Annex H (informative) Guide to use of protect directives 597

Annex I (informative) Glossary 605

Annex J (informative) Bibliography 632

Annex K (informative) Participants 633

Index 634

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Full document details: IEC 61691-1-1 2023

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 document(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.

   IEEE Standards documents are developed within IEEE Societies and Standards Coordinating Committees of the IEEE Standards Association (IEEE SA) Standards Board. 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 IEEE and serve without compensation. While IEEE administers the process and establishes rules to promote fairness in the consensus development process, IEEE does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards. Use of IEEE Standards documents is wholly voluntary. *IEEE documents are made available for use subject to important notices and legal disclaimers (see* https://standards.ieee.org/ipr/disclaimers.html *for more information).*

   IEC collaborates closely with IEEE in accordance with conditions determined by agreement between the two organizations. This Dual Logo International Standard was jointly developed by the IEC and IEEE under the terms of that agreement.

2) The formal decisions 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. The formal decisions of IEEE on technical matters, once consensus within IEEE Societies and Standards Coordinating Committees has been reached, is determined by a balanced ballot of materially interested parties who indicate interest in reviewing the proposed standard. Final approval of the IEEE standards document is given by the IEEE Standards Association (IEEE SA) Standards Board.

3) IEC/IEEE Publications have the form of recommendations for international use and are accepted by IEC National Committees/IEEE Societies in that sense. While all reasonable efforts are made to ensure that the technical content of IEC/IEEE Publications is accurate, IEC or IEEE 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 (including IEC/IEEE Publications) transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC/IEEE Publication and the corresponding national or regional publication shall be clearly indicated in the latter.

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

6) All users should ensure that they have the latest edition of this publication.

7) No liability shall attach to IEC or IEEE or their directors, employees, servants or agents including individual experts and members of technical committees and IEC National Committees, or volunteers of IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE SA) Standards Board, for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication, use of, or reliance upon, this IEC/IEEE Publication or any other IEC or IEEE Publications.

8) Attention is drawn to the normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.

9) Attention is drawn to the possibility that implementation of this IEC/IEEE Publication may require use of material 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. IEC or IEEE shall not be held responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patent Claims or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility.

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: IEC 61691-1-1 2023

IEC 61691-1-1/IEEE Std 1076 was processed through IEC technical committee 91: Electronics assembly technology, under the IEC/IEEE Dual Logo Agreement. It is an International Standard.

The text of this International Standard is based on the following documents:

IEEE Std	FDIS	Report on voting
1076 (2019)	91/1871/FDIS	91/1885/RVD

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

The language used for the development of this International Standard is English.

The IEC Technical Committee and IEEE Technical Committee have decided that the contents of this document will remain unchanged until the stability date indicated on the IEC website under webstore.iec.ch in the data related to the specific document. At this date, the document will be

- reconfirmed,
- withdrawn, or
- revised.

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: IEC 61691-1-1 2023

IEEE Std 1076™-2019
(Revision of
IEEE Std 1076-2008)

IEEE Standard for VHDL Language Reference Manual

Developed by the

Design Automation Standards Committee
of the
IEEE Computer Society

Approved 5 September 2019

IEEE SA Standards Board

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: IEC 61691-1-1 2023

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, IEEE 1076™, VHDL

Documents
Document Sharing Hub

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.

Full document details: IEC 61691-1-1 2023

Important Notices and Disclaimers Concerning IEEE Standards Documents

IEEE documents are made available for use subject to important notices and legal disclaimers. These notices and disclaimers, or a reference to this page, appear in all standards and may be found under the heading "Important Notices and Disclaimers Concerning IEEE Standards Documents." They can also be obtained on request from IEEE or viewed at http://standards.ieee.org/ipr/disclaimers.html.

Notice and Disclaimer of Liability Concerning the Use of IEEE Standards Documents

IEEE Standards documents (standards, recommended practices, and guides), both full-use and trial-use, are developed within IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association ("IEEE SA") Standards Board. IEEE ("the Institute") develops its standards through a consensus development process, approved by the American National Standards Institute ("ANSI"), which brings together volunteers representing varied viewpoints and interests to achieve the final product. IEEE Standards are documents developed through scientific, academic, and industry-based technical working groups. Volunteers in IEEE working groups are not necessarily members of the Institute and participate without compensation from IEEE. While IEEE administers the process and establishes rules to promote fairness in the consensus development process, IEEE does not independently evaluate, test, or verify the accuracy of any of the information or the soundness of any judgments contained in its standards.

IEEE Standards do not guarantee or ensure safety, security, health, or environmental protection, or ensure against interference with or from other devices or networks. Implementers and users of IEEE Standards documents are responsible for determining and complying with all appropriate safety, security, environmental, health, and interference protection practices and all applicable laws and regulations.

IEEE does not warrant or represent the accuracy or content of the material contained in its standards, and expressly disclaims all warranties (express, implied and statutory) not included in this or any other document relating to the standard, including, but not limited to, the warranties of: merchantability; fitness for a particular purpose; non-infringement; and quality, accuracy, effectiveness, currency, or completeness of material. In addition, IEEE disclaims any and all conditions relating to: results; and workmanlike effort. IEEE standards documents are supplied "AS IS" and "WITH ALL FAULTS."

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

In publishing and making its standards available, IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity nor is IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing any IEEE Standards document, should rely upon his or her own independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

IN NO EVENT SHALL IEEE BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO: PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE PUBLICATION, USE OF, OR RELIANCE UPON ANY STANDARD, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE AND REGARDLESS OF WHETHER SUCH DAMAGE WAS FORESEEABLE.

Documents
Document Sharing Hub

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.

Full document details: IEC 61691-1-1 2023

Translations

The IEEE consensus development process involves the review of documents in English only. In the event that an IEEE standard is translated, only the English version published by IEEE should be considered the approved IEEE standard.

Official statements

A statement, written or oral, that is not processed in accordance with the IEEE SA Standards Board Operations Manual shall not be considered or inferred to be the official position of IEEE or any of its committees and shall not be considered to be, or be relied upon as, a formal position of IEEE. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position of IEEE.

Comments on standards

Comments for revision of IEEE Standards documents are welcome from any interested party, regardless of membership affiliation with IEEE. However, IEEE does not provide consulting information or advice pertaining to IEEE Standards documents. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Since IEEE standards represent a consensus of concerned interests, it is important that any responses to comments and questions also receive 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 comments or questions except in those cases where the matter has previously been addressed. For the same reason, IEEE does not respond to interpretation requests. Any person who would like to participate in revisions to an IEEE standard is welcome to join the relevant IEEE working group.

Comments on standards should be submitted to the following address:

Secretary, IEEE SA Standards Board
445 Hoes Lane
Piscataway, NJ 08854 USA

Laws and regulations

Users of IEEE Standards documents should consult all applicable laws and regulations. Compliance with the provisions of any IEEE Standards document 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

IEEE draft and approved standards are copyrighted by IEEE under US and international copyright laws. They are made available by IEEE and are adopted 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 these documents available for use and adoption by public authorities and private users, IEEE does not waive any rights in copyright to the documents.

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Full document details: IEC 61691-1-1 2023

Photocopies

Subject to payment of the appropriate fee, IEEE will grant users a limited, non-exclusive license to photocopy portions of any individual standard for company or organizational internal use or individual, non-commercial use only. To arrange for payment of licensing fees, 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.

Updating of IEEE Standards documents

Users of IEEE Standards documents 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.

Every IEEE standard is subjected to review at least every 10 years. When a document is more than 10 years old and has not undergone a revision process, 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 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 IEEE Xplore at http://ieeexplore.ieee.org/ or contact IEEE at the address listed previously. For more information about the IEEE SA or IEEE's standards development process, visit the IEEE SA Website at http://standards.ieee.org.

Errata

Errata, if any, for IEEE standards can be accessed via https://standards.ieee.org/standard/index.html. Search for standard number and year of approval to access the web page of the published standard. Errata links are located under the Additional Resources Details section. Errata are also available in IEEE Xplore: https://ieeexplore.ieee.org/browse/standards/collection/ieee/. Users are encouraged to periodically check for errata.

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 by the IEEE with respect to the existence or validity of any patent rights in connection therewith. If a patent holder or patent applicant has filed a statement of assurance via an Accepted Letter of Assurance, then the statement is listed on the IEEE SA Website at https://standards.ieee.org/about/sasb/patcom/patents.html. Letters of Assurance may indicate whether the Submitter is willing or unwilling to grant licenses under patent rights without compensation or under reasonable rates, with reasonable terms and conditions that are demonstrably free of any unfair discrimination to applicants desiring to obtain such licenses.

Essential Patent Claims may exist for which a Letter of Assurance has not been received. The IEEE is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims, or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Full document details: IEC 61691-1-1 2023

IEEE Introduction

This introduction is not part of IEEE Std 1076-2019, IEEE Standard for VHDL Language Reference Manual.

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

This document, IEEE Std 1076-2019, is a revision of IEEE Std 1076-2008. The IEEE VHDL Analysis and Standardization Group (VASG), otherwise known as the 1076 Working Group, gathered the requirements, developed language extensions, and completed a draft of this revised Language Reference Manual. That draft was returned to IEEE for final revision and approval, resulting in this document. This revision incorporates numerous enhancements, both major and minor, to previously existing language features 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 and encourages the participation of all interested parties in future language revisions.[1]

Acknowledgments

Material in clause titled “Protect Tool Directives” is derived from the document titled “A Mechanism 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.

[1]If interested in participating, please contact the VASG at vhdl-200x@LISTSERV.IEEE.ORG or http://www.eda-twiki.org/cgi-bin/view.cgi/P1076/WebHome.

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Full document details: IEC 61691-1-1 2023

IEEE Standard for VHDL Language Reference Manual

1. Overview

1.1 Scope

This standard defines the syntax and semantics of the VHSIC Hardware Description Language (VHDL). The acronym VHSIC (Very High Speed Integrated Circuits) in the language's name comes from the U.S. government program that funded early work on the standard.

1.2 Purpose

VHDL is a formal notation intended for use in all phases of the creation of electronic systems. Since it is both machine and human readable, it supports the design, development, verification, synthesis, and testing of hardware designs; the communication of hardware design data; and the maintenance, modification, and procurement of hardware. This document is intended for the implementers of tools supporting the language and for advanced users of the language.

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.

Machine readable elements that are normatively referenced are available in the IEEE 1076 Open Source Repository (see Clause 2).

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Full document details: IEC 61691-1-1 2023

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

1.3.2 Syntactic description

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

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

   formal_port_list

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

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

   array

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

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

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

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

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

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

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

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

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

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

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

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

1.3.3 Semantic description

The meaning and restrictions of a particular construct are described with a set of narrative rules immediately following the syntactic productions. In these rules, an italicized term indicates the definition of that term,

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Documents
Document Sharing Hub
Full document details: IEC 61691-1-1 2023

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™-2010.[2] All PSL constructs that appear in a VHDL description shall conform to the 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 1850-2010 with the same name but without the prefix.

1.4 Word usage

The word *shall* indicates mandatory requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted (shall equals is required to).[3, 4]

The word *should* indicates that among several possibilities one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required (should equals is recommended that).

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

[3] The use of the word *must* is deprecated and cannot be used when stating mandatory requirements, *must* is used only to describe unavoidable situations.

[4] The use of *will* is deprecated and cannot be used when stating mandatory requirements, *will* is only used in statements of fact.

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Documents
Document Sharing Hub
Full document details: IEC 61691-1-1 2023

The word *may* is used to indicate a course of action permissible within the limits of the standard (may equals is permitted to).

The word *can* is used for statements of possibility and capability, whether material, physical, or causal (can equals is able to).

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: IEC 61691-1-1 2023

2. Normative references

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

IEEE 1076 Open Source Repository.[5]

IEEE Std 754™-2019, IEEE Standard for Floating-Point Arithmetic.[6, 7]

IEEE Std 1735™-2014, IEEE Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP).

IEEE Std 1850™-2010, IEEE Standard for Property Specification Language (PSL).

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

ISO/IEC 9899:2018, Information technology—Programming Languages—C.

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

[5]Jim Lewis, Patrick Lehmann, Robert Gaddi, et al., 2018. "Packages" https://opensource.ieee.org/vasg/Packages/-/archive/release/Packages-release.zip (supplement to https://opensource.ieee.org/vasg/Packages).

[6]IEEE publications are available from The Institute of Electrical and Electronics Engineers (http://standards.ieee.org).

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

[8]ISO publications are available from the International Organization for Standardization (http://www.iso.org/). IEC publications are available from the International Electrotechnical Commission (http://www.iec.ch). ISO/IEC publications are available from the American National Standards Institute (http://www.ansi.org/).

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Full document details: IEC 61691-1-1 2023

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 licence from IEEE. © 2019 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: IEC 61691-1-1 2023

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
  | mode_view_declaration

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Documents
Document Sharing Hub
Full document details: IEC 61691-1-1 2023

| constant_declaration
| signal_declaration
| *shared*_variable_declaration
| file_declaration
| alias_declaration
| attribute_declaration
| attribute_specification
| disconnection_specification
| use_clause
| group_template_declaration
| group_declaration
| *PSL*_Property_Declaration
| *PSL*_Sequence_Declaration
| *PSL*_Clock_Declaration

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

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

Example:

— An entity declaration with entity declarative items:

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

NOTE—The entity declarative part of a design entity whose corresponding architecture is decorated with the 'FOREIGN attribute is subject to special elaboration rules. See 14.4.1.[9]

3.2.4 Entity statement part

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

[9]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 licence from IEEE. © 2019 IEEE. All rights reserved.
Documents
Document Sharing Hub
Full document details: IEC 61691-1-1 2023

entity_statement_part ::=
{ entity_statement }

entity_statement ::=
concurrent_assertion_statement
| *passive*_concurrent_procedure_call_statement
| *passive*_process_statement
| *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.

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Documents
Document Sharing Hub
Full document details: IEC 61691-1-1 2023

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

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

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

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

3.3.2 Architecture declarative part

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

```
architecture_declarative_part ::=
    { block_declarative_item }

block_declarative_item ::=
        subprogram_declaration
    |   subprogram_body
    |   subprogram_instantiation_declaration
    |   package_declaration
    |   package_body
    |   package_instantiation_declaration
    |   type_declaration
    |   subtype_declaration
    |   mode_view_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.

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Full document details: IEC 61691-1-1 2023

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

3.3.3 Architecture statement part

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

architecture_statement_part ::=
{ concurrent_statement }

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

Examples:

— A body of entity Full_Adder:

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

— A body of entity TestBench:

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

— A body of entity AndGate:

```
architecture Behavior of AndGate is
begin
    process (Inputs)
        variable Temp: Bit;
    begin
        Temp := '1';
    for i in Inputs'Range loop
        if Inputs(i) = '0' then
            Temp := '0';
```

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Documents
Document Sharing Hub
Full document details: IEC 61691-1-1 2023

```
            exit;
         end if;
      end loop;
      Result <= Temp after 10 ns;
   end process;
end Behavior;
```

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

3.4 Configuration declarations

3.4.1 General

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

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

configuration_declarative_part ::=
    { configuration_declarative_item }

configuration_declarative_item ::=
    use_clause
  | attribute_specification
  | group_declaration

The entity name identifies the name of the entity declaration that defines the design entity at the root of the design hierarchy.

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.

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Documents
Document Sharing Hub
Full document details: IEC 61691-1-1 2023

Examples:

— An architecture of a microprocessor:

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

— A configuration of the microprocessor:

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

3.4.2 Block configuration

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

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

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

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Documents
Document Sharing Hub
Full document details: IEC 61691-1-1 2023

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

configuration_item ::=
    block_configuration
  | component_configuration

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

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

If a block configuration appears immediately within a component configuration, then the corresponding components shall be fully bound (see 7.3.2.2), the block specification of that block configuration shall be an architecture name, and that architecture name shall denote the same architecture body as that to which the corresponding components are bound.

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

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

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

For any name that is the label of a generate statement immediately within a given block, one or more corresponding block configurations may appear as configuration items immediately within a block configuration corresponding to the given block. Such block configurations apply to implicit blocks generated by that generate statement. If such a block configuration contains a generate specification that is a static discrete range, then the block configuration applies to those implicit block statements that are generated for the specified range of values of the corresponding generate parameter; the discrete range has no significance other than to define the set of generate statement parameter values implied by the discrete range. If such a block configuration contains a generate specification that is a static expression, then the block configuration applies only to the implicit block statement generated for the specified value of the corresponding generate parameter. If such a block configuration contains a generate specification that is an

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Documents
Document Sharing Hub
Full document details: IEC 61691-1-1 2023

alternative label, then the block configuration applies only to the implicit block generated for the generate statement body following the alternative label in the generate statement, if and only if the condition after the alternative label evaluates to TRUE (for an if generate statement) or the case generate alternative containing the alternative label is the chosen alternative (for a case generate statement). If no generate specification appears in such a block configuration, then it applies to exactly one of the following sets of blocks:

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

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

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

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

— If the generate specification is a discrete range or an expression, then it is an error if the generate statement denoted by the generate statement label is not a for generate statement. Moreover, for a generate specification that is a discrete range, it is an error if the type of the discrete range is not the same as the type of the discrete range of the generate parameter specification and if any value in the range does not belong to the discrete range of the generate parameter specification. Similarly, for a generate specification that is an expression, it is an error if the type of the expression is not the same as the type of the discrete range of the generate parameter specification and if the value of the expression does not belong to the discrete range of the generate parameter specification.

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

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

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

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

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

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Full document details: IEC 61691-1-1 2023

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 will denote one of these architectures.

Examples:

— A block configuration for a design entity:

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

— A block configuration for a block statement:

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

3.4.3 Component configuration

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

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

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

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

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

If a given component instance is unbound in the corresponding block, then any explicit component configuration for that instance that does not contain an explicit binding indication will contain an implicit,

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Documents
Document Sharing Hub
Full document details: IEC 61691-1-1 2023

default binding indication (see 7.3.3). Similarly, if a given component instance is unbound in the corresponding block, then any implicit component configuration for that instance will contain an implicit, default binding indication.

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

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

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

Examples:

— A component configuration with binding indication:

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

— A component configuration containing block configurations:

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

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

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


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

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Documents
Document Sharing Hub
Full document details: IEC 61691-1-1 2023

```
                ...
            end for;      -- X
        end for; -- C
    end for; -- A
end configuration Illegal;
```

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: IEC 61691-1-1 2023

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. For the purposes of determining purity, the return identifier of a function is considered a parameter. 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** [*return*_identifier **of**] type_mark

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

    [generic_map_aspect]]

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Documents
Document Sharing Hub
Full document details: IEC 61691-1-1 2023

designator ::= identifier | operator_symbol

operator_symbol ::= string_literal

The specification of a procedure specifies its designator, its generics (if any), and its formal parameters (if any). The specification of a function specifies its designator, its generics (if any), its formal parameters (if any), its return identifier (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.

For a function that includes a return identifier, an implicit subtype declaration, whose name is the return identifier, is created as the first declaration in the subprogram declarative part. The properties of the implicitly declared subtype denoted by the return identifier shall be determined from the immediate context of the function call, as follows:

a) If the function call is the expression following the assignment symbol ":=" in a constant, variable, or signal declaration, then the subtype denoted by the return identifier is the subtype of the corresponding object declaration.

b) If the function call is the value expression in an assignment statement whose target is not an aggregate, then the subtype denoted by the return identifier is the subtype of the target of the assignment statement.

c) If the function call is the actual in an association element, then the subtype denoted by the return identifier is the subtype of the corresponding formal.

d) If the function call is the formal in an association element, then the subtype denoted by the return identifier is the subtype of the corresponding actual.

e) If the function call is the operand in a qualified expression, then the subtype denoted by the return identifier is the subtype defined by the type mark of the qualified expression.

It is an error if a function has a return identifier and

— the function is called from a context other than the ones above or

— the subtype of the return identifier is a unconstrained or partially constrained composite subtype.

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 without a generic map aspect 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.

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: IEC 61691-1-1 2023

It is an error if the result subtype of a function denotes either a file type, a protected type, or a composite type with a subelement of 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 will not directly reference variables declared immediately within the declarative region associated with the protected type. However, impure functions and procedures whose bodies appear in the protected type body may make such references. Such references are made only when the referencing subprogram has exclusive access to the declarative region associated with the protected type.

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

4.2.2 Formal parameters

4.2.2.1 Formal parameter lists

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

formal_parameter_list ::= *parameter*_interface_list

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

For those parameters with modes, the only modes that are allowed for formal parameters of a procedure or an impure function 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 pure 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 1—Variable class parameters of access and protected types are allowed for procedures and impure functions.

NOTE 2—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.

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Documents
Document Sharing Hub
Full document details: IEC 61691-1-1 2023

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 non-foreign 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 non-foreign 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.

If a subprogram has a parameter whose type is a protected type or a composite type with a subelement of a protected type, then the parameter shall be passed by reference and the mode shall be **inout**.

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

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

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: IEC 61691-1-1 2023

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

4.2.2.3 Signal parameters

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

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

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

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

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

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

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

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

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

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

4.2.2.4 File parameters

For parameters of class **file**, references to the actual file are passed into the subprogram. No particular parameter-passing mechanism is defined by the language, but a reference to the formal parameter shall be

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Documents
Document Sharing Hub
Full document details: IEC 61691-1-1 2023

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 or a composite type with a subelement 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. Similarly, it is an error if a formal of a composite type with a subelement of a file type is associated with an actual that is not of a composite type with a subelement of a file type.

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

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

4.3 Subprogram bodies

A subprogram body specifies the execution of a subprogram.

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

subprogram_declarative_part ::=
    { subprogram_declarative_item }

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

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: IEC 61691-1-1 2023

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 subprogram declarative part declares a shared variable (see 6.4.2.4).

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

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

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

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

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

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Full document details: IEC 61691-1-1 2023

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 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 will be given if a call of the subprogram occurs textually before the subprogram body, and that such a declaration will occur before the call itself.

NOTE 2—The preceding rules concerning pure function subprograms, together with the fact that pure function parameters are of mode **in**, imply that a pure function has no effect other than the computation of the returned value. Thus, a pure function with static inputs may be invoked either during elaboration or simulation without impacting 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 will 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 will not have a parent that is a subprogram declared within either a protected type declaration or a protected type body.

4.4 Subprogram instantiation declarations

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

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Documents
Document Sharing Hub
Full document details: IEC 61691-1-1 2023

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 either have the same base type, or one or both have an unspecified type indication. Two subprograms are said to have the same *parameter and result type profile* if and only if both have the same parameter type profile, and if either both are functions with the same result base type or neither of the two is a function.

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

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Documents
Document Sharing Hub
Full document details: IEC 61691-1-1 2023

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 8.4). Further, ambiguities may (but need not) arise when the declarations of overloaded subprograms in an uninstantiated declaration have parameter and result type profiles that involve different formal generic types of the uninstantiated declaration. If the declaration is instantiated with the same actual type associated with the formals, the resulting overloaded subprograms in the instance may have the same parameter and result type profile. Such ambiguities cannot be solved.

4.5.2 Operator overloading

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

The subprogram specification of a unary operator shall have a single parameter, unless the subprogram specification is a method (see 5.6.2) of a protected type. In this latter case, the subprogram specification shall have no parameters. The subprogram specification of a binary operator shall have two parameters, unless the subprogram specification is a method of a protected type, in which case, the subprogram specification shall have a single parameter. If the subprogram specification of a binary operator has two parameters, for each use of this operator, the first parameter is associated with the left operand, and the second parameter is associated with the right operand.

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Full document details: IEC 61691-1-1 2023

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, generic map aspect, 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 either the same as the base type of the corresponding formal parameter of the subprogram or the subtype indication of the formal parameter is an unspecified type indication.

— 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.

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Full document details: IEC 61691-1-1 2023

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
-- Clause 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 8.7). 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
```

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Documents
Document Sharing Hub
Full document details: IEC 61691-1-1 2023

```
            return '1';
          end if;
        end loop;
        return '0';
      end if;
    end function WIRED_OR;
```

4.7 Package declarations

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

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

package_header ::=
  [generic_clause
  [generic_map_aspect ;]]

package_declarative_part ::=
  { package_declarative_item }

package_declarative_item ::=
    subprogram_declaration
  | subprogram_instantiation_declaration
  | package_declaration
  | package_instantiation_declaration
  | type_declaration
  | subtype_declaration
  | mode_view_declaration
  | constant_declaration
  | signal_declaration
  | shared_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

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Documents
Document Sharing Hub
Full document details: IEC 61691-1-1 2023

uninstantiated package. If the package header contains both a generic clause and a generic map aspect, the package is called a *generic-mapped package*. A package declared with a generic clause in which every generic declaration has a default, and with no generic map aspect, is considered to be an uninstantiated package, not a generic-mapped package with default associations for all of the generic declarations.

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

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

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

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: IEC 61691-1-1 2023

package_body_declarative_part ::=
    { package_body_declarative_item }

package_body_declarative_item ::=
        subprogram_declaration
    | subprogram_body
    | subprogram_instantiation_declaration
    | package_declaration
    | package_body
    | package_instantiation_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | shared_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 shared variable is declared in the package declarative part.

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;
```

Published by IEC under licence from IEEE. © 2019 IEEE. All rights reserved.
Documents
Document Sharing Hub
Full document details: IEC 61691-1-1 2023