IEEE Standards

IEEE Std 1076.6™-2004
(Revision of
IEEE Std 1076.6-1999)

1076.6™

IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

IEEE

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

11 October 2004

Print: SH95242
PDF: SS95242

Recognized as an
American National Standard (ANSI)

IEEE Std 1076.6™-2004
(Revision of
IEEE Std 1076.6-1999)

IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society

Approved 25 August 2004

American National Standard Institute

Approved 12 May 2004

IEEE-SA Standards Board

Abstract: This document specifies a standard for use of very high-speed integrated circuit hardware description language (VHDL) to model synthesizable register-transfer level digital logic. A standard syntax and semantics for VHDL register-transfer level synthesis is defined. The subset of the VHDL language, which is synthesizable, is described, and nonsynthesizable VHDL constructs are identified that should be ignored or flagged as errors.
Keywords: hardware description language, logic synthesis, register transfer level (RTL), very high-speed integrated circuit hardware description language (VHDL)

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

Print: ISBN 0-7381-4064-3 SH95242
PDF: ISBN 0-7381-4065-1 SS95242

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

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

The IEEE does not warrant or represent the accuracy or content of the material contained herein, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or fitness for a specific purpose, or that the use of the material contained herein is free from patent infringement. IEEE Standards documents are supplied "**AS IS**."

The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

In publishing and making this document available, the IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is the IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEEE Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position, explanation, or interpretation of the IEEE.

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

Secretary, IEEE-SA Standards Board

445 Hoes Lane

P.O. Box 1331

Piscataway, NJ 08855-1331USA

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

Introduction

(This introduction is not part of IEEE Std 1076.6-2004, IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis.)

This standard describes a standard syntax and semantics for VHDL RTL synthesis. It defines the subset of IEEE Std 1076™-2002 (VHDL) that is suitable for RTL synthesis and defines the semantics of that subset for the synthesis domain. This standard is based on IEEE Std 1076-2002, IEEE Std 1164™-1993, and IEEE Std 1076.3™-1997.

The purpose of this standard is to define a syntax and semantics that can be used in common by all compliant RTL synthesis tools to achieve uniformity of results in a similar manner to which simulation tools use IEEE Std 1076-2002. This will allow users of synthesis tools to produce well-defined designs whose functional characteristics are independent of a particular synthesis implementation by making their designs compliant with this standard.

The standard is intended for use by logic designers and electronic engineers.

This document specifies IEEE Std 1076.6-2004, which is a revision of IEEE Std 1076.6-1999. The VHDL Synthesis Interoperability Working Group (SIWG) of the IEEE Computer Society started the development of IEEE Std 1076.6-2004 in January 1998. The work initially started as a Level 2 effort (Level 1 being IEEE Std 1076.6-1999). In fact the work on Level 2 continued right after Level 1 was completed by the working group. The working group realized that a Level 2 was required and that it would take some time to develop and continued working on it at regular face-to-face meetings and teleconferences. As the Level 2 draft continued to mature, the working group decided that rather than having two different levels of synthesis subsets, it was better to just have one standard, with IEEE Std 1076.6-2004 becoming Level 2.

The intent of this version was to include a maximum subset of VHDL that could be used to describe synthesizable RTL logic. This included considering new features introduced by IEEE Std 1076-2002, new semantics based on algorithmic styles rather than template-driven, and a set of synthesis attributes that could be used to annotate an RTL description. The following team leaders drove this effort:

Syntax: Lance Thompson

Semantics: Vinaya Singh

Attributes: Sanjiv Narayan

In addition, the following provided much-needed additional support:

Web and reflector admin: David Bishop

Documentation: John Michael Williams

A majority of the work conducted by the working group was done via teleconferencing, which was held regularly and open to all. Also, the working group used an e-mail reflector and its web page effectively to distribute and share information.

The following volunteers contributed to the development of this standard:

J. Bhasker, *Chair*

Jim Lewis, *Vice-Chair*

Rob Anderson	Rich Hatcher	Jonas Nilsson
Bill Anker	Mohammad Kakoee	Alain Raynaud
Victor Berman	Masamichi Kawarabayashi	Mehrdad Reshadi
David Bishop	Apurva Kalia	Fredj Rouatbi
Dominique Borrione	Satish Kumar	Steve Schultz
Dennis Brophy	Evan Lavelle	Manish Shrivastava
Andrew Brown	Vijay Madisetti	Vinaya Singh
Patrick Bryant	Erich Marschner	Douglas Smith
Ben Cohen	Paul Menchini	Lance Thompson
Tim Davis	Amitabh Menon	Alessandro Uber
Colin Dente	Egbert Molenkamp	Jim Vellenga
Wolfgang Ecker	Bob Myers	Eugenio Villar
Bob Flatt	Sanjana Nair	John Michael Williams
Christopher Grimm	Sanjiv Narayan	Francisco De Ycaza
Steve Grout	Zain Navabi	Alex Zamfirescu

Development of IEEE Std 1076.6-1999

Initial work on this standard started as a synthesis interoperability working group under VHDL International. The working group was also chartered by the EDA Industry Council Project Technical Advisory Board (PTAB) to develop a draft based on the donated subsets by the following companies/groups:

- Cadence
- European Synthesis Working Group
- IBM
- Mentor Graphics
- Synopsys

After the PTAB approved of the draft 1.5 with an overwhelming affirmative response, an IEEE PAR was obtained to clear its way for IEEE standardization. Most of the members of the original group continued to be part of the Pilot Group under P1076.6 to lead the technical work.

At the time the 1999 standard was completed, the P1076.6 Pilot Team had the following membership:

Rob Anderson	Wolfgang Ecker	Doug Perry
Victor Berman	Bob Flatt	Steve Schultz
J. Bhasker	Christopher Grimm	Doug Smith
David Bishop	Rich Hatcher	Lance Thompson
Dominique Borrione	Apurva Kalia	Fur-Shing Tsai
Dennis Brophy	Masamichi Kawarabayashi	Jim Vellenga
Ben Cohen	Jim Lewis	Eugenio Villar
Colin Dente	Sanjiv Narayan	Nels Vander Zanden

Many individuals from different organizations contributed to the development of this standard. In particular, in addition to the Pilot Team, the following individuals contributed to the development of the standard by being part of the working group:

Bill Anker	Robert Blackburn	John Hillawi
LaNae Avra		Pradip Jha

In addition, 95 individuals on the working group e-mail reflector also contributed to this development.

Notice to users

Errata

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

Interpretations

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

Patents

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

Participants

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

Bill Anker
Peter Ashenden
John Aynsley
Stephen Bailey
Jayaram Bhasker
Stefen Boyd
Kai Moon Chow
Keith Chow
Guru Dutt Dhingra
Colin Dente
George Economakos
Peter Flake
Ian Andrew Guyler
William A. Hanna
Jim Lewis
Michael McNamara
D. C. Mohla
E. Molenkamp
Serafin A. Perez Lopez
John Shields
Mark Tillinghast
John Michael Williams
Mark Zwolinski

When the IEEE-SA Standards Board approved this standard on 12 May 2004, it had the following membership:

Contents

IEEE Standards

IEEE Std 1076.6™-2004
(Revision of
IEEE Std 1076.6-1999)

1076.6™

IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

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

11 October 2004

Print: SH95242
PDF: SS95242

Recognized as an
American National Standard (ANSI)

IEEE Std 1076.6™-2004
(Revision of
IEEE Std 1076.6-1999)

IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society

Approved 25 August 2004

American National Standard Institute

Approved 12 May 2004

IEEE-SA Standards Board

Abstract: This document specifies a standard for use of very high-speed integrated circuit hardware description language (VHDL) to model synthesizable register-transfer level digital logic. A standard syntax and semantics for VHDL register-transfer level synthesis is defined. The subset of the VHDL language, which is synthesizable, is described, and nonsynthesizable VHDL constructs are identified that should be ignored or flagged as errors.
Keywords: hardware description language, logic synthesis, register transfer level (RTL), very high-speed integrated circuit hardware description language (VHDL)

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

Print: ISBN 0-7381-4064-3 SH95242
PDF: ISBN 0-7381-4065-1 SS95242

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

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

The IEEE does not warrant or represent the accuracy or content of the material contained herein, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or fitness for a specific purpose, or that the use of the material contained herein is free from patent infringement. IEEE Standards documents are supplied "**AS IS**."

The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

In publishing and making this document available, the IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is the IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEEE Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position, explanation, or interpretation of the IEEE.

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

Secretary, IEEE-SA Standards Board

445 Hoes Lane

P.O. Box 1331

Piscataway, NJ 08855-1331USA

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

Introduction

(This introduction is not part of IEEE Std 1076.6-2004, IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis.)

This standard describes a standard syntax and semantics for VHDL RTL synthesis. It defines the subset of IEEE Std 1076™-2002 (VHDL) that is suitable for RTL synthesis and defines the semantics of that subset for the synthesis domain. This standard is based on IEEE Std 1076-2002, IEEE Std 1164™-1993, and IEEE Std 1076.3™-1997.

The purpose of this standard is to define a syntax and semantics that can be used in common by all compliant RTL synthesis tools to achieve uniformity of results in a similar manner to which simulation tools use IEEE Std 1076-2002. This will allow users of synthesis tools to produce well-defined designs whose functional characteristics are independent of a particular synthesis implementation by making their designs compliant with this standard.

The standard is intended for use by logic designers and electronic engineers.

This document specifies IEEE Std 1076.6-2004, which is a revision of IEEE Std 1076.6-1999. The VHDL Synthesis Interoperability Working Group (SIWG) of the IEEE Computer Society started the development of IEEE Std 1076.6-2004 in January 1998. The work initially started as a Level 2 effort (Level 1 being IEEE Std 1076.6-1999). In fact the work on Level 2 continued right after Level 1 was completed by the working group. The working group realized that a Level 2 was required and that it would take some time to develop and continued working on it at regular face-to-face meetings and teleconferences. As the Level 2 draft continued to mature, the working group decided that rather than having two different levels of synthesis subsets, it was better to just have one standard, with IEEE Std 1076.6-2004 becoming Level 2.

The intent of this version was to include a maximum subset of VHDL that could be used to describe synthesizable RTL logic. This included considering new features introduced by IEEE Std 1076-2002, new semantics based on algorithmic styles rather than template-driven, and a set of synthesis attributes that could be used to annotate an RTL description. The following team leaders drove this effort:

Syntax: Lance Thompson

Semantics: Vinaya Singh

Attributes: Sanjiv Narayan

In addition, the following provided much-needed additional support:

Web and reflector admin: David Bishop

Documentation: John Michael Williams

A majority of the work conducted by the working group was done via teleconferencing, which was held regularly and open to all. Also, the working group used an e-mail reflector and its web page effectively to distribute and share information.

The following volunteers contributed to the development of this standard:

J. Bhasker, *Chair*

Jim Lewis, *Vice-Chair*

Rob Anderson
Bill Anker
Victor Berman
David Bishop
Dominique Borrione
Dennis Brophy
Andrew Brown
Patrick Bryant
Ben Cohen
Tim Davis
Colin Dente
Wolfgang Ecker
Bob Flatt
Christopher Grimm
Steve Grout
Rich Hatcher
Mohammad Kakoee
Masamichi Kawarabayashi
Apurva Kalia
Satish Kumar
Evan Lavelle
Vijay Madisetti
Erich Marschner
Paul Menchini
Amitabh Menon
Egbert Molenkamp
Bob Myers
Sanjana Nair
Sanjiv Narayan
Zain Navabi
Jonas Nilsson
Alain Raynaud
Mehrdad Reshadi
Fredj Rouatbi
Steve Schultz
Manish Shrivastava
Vinaya Singh
Douglas Smith
Lance Thompson
Alessandro Uber
Jim Vellenga
Eugenio Villar
John Michael Williams
Francisco De Ycaza
Alex Zamfirescu

Development of IEEE Std 1076.6-1999

Initial work on this standard started as a synthesis interoperability working group under VHDL International. The working group was also chartered by the EDA Industry Council Project Technical Advisory Board (PTAB) to develop a draft based on the donated subsets by the following companies/groups:

— Cadence
— European Synthesis Working Group
— IBM
— Mentor Graphics
— Synopsys

After the PTAB approved of the draft 1.5 with an overwhelming affirmative response, an IEEE PAR was obtained to clear its way for IEEE standardization. Most of the members of the original group continued to be part of the Pilot Group under P1076.6 to lead the technical work.

At the time the 1999 standard was completed, the P1076.6 Pilot Team had the following membership:

Rob Anderson
Victor Berman
J. Bhasker
David Bishop
Dominique Borrione
Dennis Brophy
Ben Cohen
Colin Dente
Wolfgang Ecker
Bob Flatt
Christopher Grimm
Rich Hatcher
Apurva Kalia
Masamichi Kawarabayashi
Jim Lewis
Sanjiv Narayan
Doug Perry
Steve Schultz
Doug Smith
Lance Thompson
Fur-Shing Tsai
Jim Vellenga
Eugenio Villar
Nels Vander Zanden

Many individuals from different organizations contributed to the development of this standard. In particular, in addition to the Pilot Team, the following individuals contributed to the development of the standard by being part of the working group:

Bill Anker
LaNae Avra
Robert Blackburn
John Hillawi
Pradip Jha

In addition, 95 individuals on the working group e-mail reflector also contributed to this development.

Notice to users

Errata

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

Interpretations

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

Patents

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

Participants

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

Bill Anker
Peter Ashenden
John Aynsley
Stephen Bailey
Jayaram Bhasker
Stefen Boyd
Kai Moon Chow
Keith Chow
Guru Dutt Dhingra
Colin Dente
George Economakos
Peter Flake
Ian Andrew Guyler
William A. Hanna
Jim Lewis
Michael McNamara
D. C. Mohla
E. Molenkamp
Serafin A. Perez Lopez
John Shields
Mark Tillinghast
John Michael Williams
Mark Zwolinski

When the IEEE-SA Standards Board approved this standard on 12 May 2004, it had the following membership:

Don Wright, *Chair*
Steve M. Mills, *Vice Chair*
Judith Gorman, *Secretary*

Chuck Adams
H. Stephen Berger
Mark D. Bowman
Joseph A. Bruder
Bob Davis
Roberto de Boisson
Julian Forster*
Arnold M. Greenspan
Mark S. Halpin
Raymond Hapeman
Richard J. Holleman
Richard H. Hulett
Lowell G. Johnson
Joseph L. Koepfinger*
Hermann Koch
Thomas J. McGean
Daleep C. Mohla
Paul Nikolich
T. W. Olsen
Ronald C. Petersen
Gary S. Robinson
Frank Stone
Malcolm V. Thaden
Doug Topping
Joe D. Watson

*Member Emeritus

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

Satish K. Aggarwal, *NRC Representative*
Richard DeBlasio, *DOE Representative*
Alan Cookson, *NIST Representative*

Don Messina
IEEE Standards Project Editor

Contents

IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis

1. Overview

1.1 Scope

This standard defines a subset of very high-speed integrated circuit hardware description language (VHDL) that ensures portability of VHDL descriptions between register transfer level synthesis tools. Synthesis tools may be compliant and yet have features beyond those required by this standard. This standard defines how the semantics of VHDL shall be used, for example, to model level-sensitive and edge-sensitive logic. It also describes the syntax of the language with reference to what shall be supported and what shall not be supported for interoperability.

Use of this standard should minimize the potential for functional simulation mismatches between models before they are synthesized and after they are synthesized.

1.2 Compliance to this standard

1.2.1 Model compliance

A VHDL model shall be defined as being compliant to this standard if the model

a) Uses only constructs described as supported or ignored in this standard

b) Adheres to the semantics defined in this standard

1.2.2 Tool compliance

A synthesis tool shall be defined as being compliant to this standard if it

a) Accepts all models that adhere to the model compliance definition defined in 1.2.1

b) Supports language related pragmas defined by this standard

c) Produces a circuit model that has the same functionality as the input model based on the verification process as outlined in Clause 5.

1.3 Terminology

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*). The word *should* is used to indicate that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain course of action is deprecated but not prohibited (*should* equals *is recommended that*). The word *may* indicates a course of action permissible within the limits of the standard (*may* equals *is permitted*).

A synthesis tool is said to *accept* a VHDL construct if it allows that construct to be legal input; it is said to *interpret* the construct (or to provide an *interpretation* of the construct) by producing something that represents the construct. A synthesis tool is not required to provide an interpretation for every construct that it accepts, but only for those for which an interpretation is specified by this standard.

The constructs in the standard shall be categorized as follows:

Supported: RTL synthesis shall interpret a construct, that is, map the construct to an equivalent hardware representation.

Ignored: RTL synthesis shall ignore the construct and produce a warning. Encountering the construct shall not cause synthesis to fail, but synthesis results may not match simulation results. The mechanism, if any, by which RTL synthesis notifies (warns) the user of such constructs is not defined by this standard. Ignored constructs may include unsupported constructs.

Not Supported: RTL synthesis does not support the construct. RTL synthesis does not expect to encounter the construct, and the failure mode shall be undefined. RTL synthesis may fail upon encountering such a construct. Failure is not mandatory; more specifically, RTL synthesis is allowed to treat such a construct as ignored.

NOTE—A synthesis tool may interpret constructs that are identified as not supported in this standard. However a model that contains such unsupported constructs is not compliant with this standard.[1]

1.4 Conventions

This standard uses the following conventions:

a) The body of the text of this standard uses **boldface** to denote VHDL reserved words (such as **downto**).

b) The text of the VHDL examples and code fragments is represented in a fixed-width font.

c) Syntax text that is struck-through (e.g., ~~text~~) refers to syntax that shall not be supported.

d) Syntax text that is underscored (e.g., text) refers to syntax that shall be ignored.

e) < and > pairs are used to represent text in one of several different, but specific forms. For example, one of the forms of <clock_edge> could be "CLOCK'EVENT and CLOCK = '1'".

f) Any paragraph starting with "NOTE—" is informative and not part of the standard.

g) The examples that appear in this document under "*Example:*" are for the sole purpose of demonstrating the syntax and semantics of VHDL for synthesis. It is not the intent of this standard to demonstrate, recommend, or emphasize coding styles that are more (or less) efficient in generating an equivalent hardware representation. In addition, it is not the intent of this standard to present examples that represent a compliance test suite, or a performance benchmark, even though these examples are compliant to this standard (except as noted otherwise).

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

2. References

This standard shall be used in conjunction with the following publications. When the following standards are superseded by an approved revision, the revision shall apply.

IEEE Std 1076™-2002, IEEE Standard VHDL Language Reference Manual.[2, 3]

IEEE Std 1076.3™-1997, IEEE Standard Synthesis Packages (NUMERIC_BIT and NUMERIC_STD).

IEEE Std 1164™-1993, IEEE Standard Multivalue Logic System for VHDL Model Interoperability (STD_LOGIC_1164).

3. Definitions and acronyms

3.1 Definitions

For the purposes of this standard, the following terms and definitions apply. *The Authoritative Dictionary of IEEE Standards Terms, Seventh Edition* should be referenced for terms not defined in this clause. Terms used within this standard but not defined in this clause are assumed to be from IEEE Std 1076-2002, IEEE Std 1164-1993, or IEEE Std 1076.3-1997.[4]

3.1.1 assignment reference: The occurrence of a literal or expression as the waveform element of a signal assignment statement or as the right-hand side expression of a variable assignment statement.

3.1.2 combinational logic: Logic that settles to a state entirely determined by the current input values and therefore that cannot store information. Any change in the input causes a new state completely defined by the new inputs.

3.1.3 don’t care value: The enumeration literal ‘-’ of the type STD_ULOGIC (or subtype STD_LOGIC).

3.1.4 edge-sensitive storage element: Any storage element mapped to by a synthesis tool that

a) Propagates the value at the data input whenever an appropriate transition in value is detected on a clock control input

b) Preserves the last value propagated at all other times, except when any asynchronous control inputs become active (for example, a flip-flop)

3.1.5 high-impedance value: The enumeration literal ‘Z’ of the type STD_ULOGIC (or subtype STD_LOGIC).

3.1.6 level-sensitive storage element: Any storage element mapped to by a synthesis tool that

a) Propagates the value at the data input whenever an appropriate value is detected on a clock control input

b) Preserves the last value propagated at all other times, except when any asynchronous control inputs become active (for example, a latch)

---

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

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

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

3.1.7 logical operation: An operation for which the VHDL operator is **and**, **or**, **nand**, **nor**, **xor**, **xnor**, or **not**.

3.1.8 metacomment: A VHDL comment (--) that is used to provide synthesis-specific interpretation by a synthesis tool.

3.1.9 metalogical value: One of the enumeration literals ‘U’, ‘X’, ‘W’, or ‘-’ of the type STD_ULOGIC (or subtype STD_LOGIC).

3.1.10 pragma: A generic term used to define a construct with no predefined language semantics that influences how a synthesis tool will synthesize VHDL code into an equivalent hardware representation.

3.1.11 sequential logic: Logic that settles to a state not determined solely by current inputs. The current state of such logic can be determined only by knowing the current inputs and some history of past inputs in their sequential order. Sequential logic always stores information from past input and therefore may be used to implement storage elements.

3.1.12 synchronous assignment: An assignment that takes place when a signal or variable value is updated as a direct result of a clock edge expression evaluating as true.

3.1.13 synthesis library: A library of digital design objects such as logic gates, chip pads, memory blocks, or other blocks; instances of these elements are connected together by a synthesis tool to create a synthesized netlist.

3.1.14 synthesis tool: Any system, process, or tool that interprets register transfer level VHDL source code as a description of an electronic circuit and derives a netlist description of that circuit.

3.1.15 synthesis-specific attribute: An attribute recognized by a tool compliant to this standard.

3.1.16 user: A person, system, process, or tool that generates the VHDL source code that a synthesis tool processes.

3.1.17 vector: A one-dimensional array.

3.1.18 well-defined: Containing no metalogical or high-impedance value.

3.2 Acronyms

LRM The IEEE VHDL language reference manual, that is, IEEE Std 1076-2002.

RTL The register transfer level of modeling circuits in VHDL for use with register transfer level synthesis. Register transfer level is a level of description of a digital design in which the clocked behavior of the design is expressly described in terms of data transfers between storage elements in sequential logic, which may be implied, and combinational logic, which may represent any computing or arithmetic-logic-unit logic. RTL modeling allows design hierarchy that represents a structural description of other RTL models.

4. Predefined types

A synthesis tool, compliant with this standard, shall support the following predefined types:

a) BIT, BOOLEAN, and BIT_VECTOR as defined by IEEE Std 1076-2002

b) CHARACTER and STRING as defined in IEEE Std 1076-2002

c) INTEGER as defined in IEEE Std 1076-2002

d) STD_ULOGIC, STD_ULOGIC_VECTOR, STD_LOGIC, and STD_LOGIC_VECTOR as defined by the package STD_LOGIC_1164 (IEEE Std 1164-1993)

e) SIGNED and UNSIGNED as defined by the VHDL package NUMERIC_BIT as part of IEEE Std 1076.3-1997

f) SIGNED and UNSIGNED as defined by the VHDL package NUMERIC_STD as part of IEEE Std 1076.3-1997

The synthesis interpretation of the values that belong to type STD_ULOGIC shall be as defined in IEEE Std 1076.3-1997.

No array type, other than those listed in e) and f), shall be used to represent signed or unsigned numbers.

The synthesis tool shall also support user-defined and other types derived from the predefined types according to the rules of 8.3.

By definition, if a type with a metalogical or high-impedance value is used in a model, then this type shall have as an ancestor a type that belongs to the package STD_LOGIC_1164 (IEEE Std 1164-1993).

5. Verification methodology

Synthesized results may be broadly classified as either combinational or sequential. Sequential logic has some form of internal storage (latch, register, memory). Combinational logic has outputs that are solely a function of the inputs with no internal loops and no internal storage. Designs may contain both sequential and combinational parts.

The process of verifying synthesis results using simulation consists of applying equivalent inputs to both the original model and synthesized model and then comparing their outputs to ensure that they are equivalent. Equivalent in this context means that a synthesis tool shall produce a circuit that is equivalent at the input, output, and bidirectional ports of the model. As synthesis in general does not recognize the same delays as simulators, the outputs cannot be compared at every simulation time. Rather, they can only be compared at specific simulation times when all transient delays have settled and all active timeout clauses have been exceeded. If the outputs do not match at all comparable times, the synthesis tool shall not be compliant. There shall be no matching requirement placed on any internal nodes.

The input stimulus shall comply with the following criteria:

a) Input data does not contain metalogical or high-impedance values.

b) Input data may only contain ‘H’ and ‘L’ on inputs that are converted to ‘1’ and ‘0’, respectively.

c) For combinational verification, input data must change far enough in advance of sensing times to allow transient delays to have settled.

d) Clock and/or input data must change after enough time of the asynchronous set/reset signals going from active to inactive to fulfill the setup/hold times of the sequential elements in the design.

e) For edge-sensitive designs, primary inputs of the design must change far enough in advance for the edge-sensitive storage element input data to fulfill the setup times with respect to the active clock edge. Also, the input data must remain stable for long enough to fulfill the hold times with respect to the active clock edge.

f) For level-sensitive designs, primary inputs of the design must change far enough in advance for the level-sensitive storage element input data to fulfill the setup times. Also, the input data must remain stable for long enough to fulfill the hold times.

NOTE—A synthesis tool may define metalogical or high-impedance values appearing on primary outputs in one model as equivalent to logical values in the other model. For this reason, the input stimulus may need to reset internal storage elements to specific logical values before the outputs of both models are compared for logical values.

5.1 Combinational verification

To verify combinational logic, the input stimulus shall be applied first. Sufficient time shall be provided for the design to settle, and then the outputs examined. To verify the combinational logic portion of a model, the following sequence of events shall be done repeatedly for each input stimulus application:

a) Apply input stimulus

b) Wait for data to settle

c) Check outputs

Each application of inputs shall include enough delay so that the transient delays and timeout clause delays have been exceeded. A model is not in compliance with this standard if it is possible for outputs or internal nodes of the combinational model never to reach a steady state (i.e., oscillatory behavior).

Example:

```
A <= not A after 5 ns; -- oscillatory behavior, noncompliant
```

5.2 Sequential verification

The general scheme consists of applying inputs periodically and then comparing the outputs just before the next set of inputs is applied. Sequential models contain edge-sensitive and/or level-sensitive storage elements. The sequential design must be reset, if required, before verification can begin.

The verification of designs containing edge-sensitive or level-sensitive storage elements is as follows:

a) Edge-sensitive models: The same sequence of tasks as used for combinatorial verification shall be performed during verification: Change the inputs, compute the results, and compare the outputs. However, for sequential verification, these tasks shall be synchronized with one of the inputs, which is a clock. The inputs must change in an appropriate order with respect to the input that is treated as a clock, and their consequences must be allowed to settle prior to comparison. Comparison might best be done just before the active clock edge, and the non-clock inputs can change relatively soon after the edge. The circuit then has the rest of the clock period to compute the new results before they are stored at the next clock edge. The period of the clock generated by the stimulus shall be sufficient to allow the input and output signals to settle.

b) Level-sensitive models: These designs are generally less predictable than edge-sensitive models due to the asynchronous nature of the signal interactions. Verification of synthesized results depends on the application. With level-sensitive storage elements, a general rule is that data inputs should be stable before enables go inactive (i.e., latch) and comparing of outputs is best done after enables are inactive (i.e., latched) and combinational delays have settled. A level-sensitive model in which it is possible, in the absence of further changes to the inputs of the model, for one or more internal values or outputs of the model never to reach a steady state (oscillatory behavior) is not in compliance with this standard.

6. Modeling hardware elements

This clause specifies styles for modeling hardware elements such as edge-sensitive storage elements, level-sensitive storage elements, three-state elements, and combinational elements.

This clause does not limit the optimizations that can be performed on a VHDL model. The scope of optimizations that may be performed by a synthesis tool depends on the tool itself. The hardware modeling styles specified in this clause do not take into account any optimizations or transformations. A specific tool may perform optimizations; this may result in removal of redundant or unused logic from the final netlist. This shall NOT be taken as a violation of this standard provided the synthesized netlist has the same functionality as the input model, as characterized in Clause 5.

6.1 Edge-sensitive sequential logic

6.1.1 Clock signal type

The allowed types for clock signals shall be BIT, STD_ULOGIC and their subtypes (e.g., STD_LOGIC). Only the values ‘0’ and ‘1’ from these types shall be used in expressions representing clock levels and clock edges (see 6.1.2).

Scalar elements of arrays of the above types shall be supported as clock signals.

Example:

```
signal BUS8: std_logic_vector(7 downto 0);
...
process (BUS8(0))
begin
  if BUS8(0) = '1' and BUS8(0)'EVENT then
  ...
...
-- BUS8(0) is a scalar element used as a clock signal.
```

6.1.2 Clock edge specification

The general syntax for specifying an edge of a clock shall be the following:

```
clock_edge ::=
   RISING_EDGE(clk_signal_name)
 | FALLING_EDGE(clk_signal_name)
 | clock_level and event_expr
 | event_expr  and clock_level
 clock_level ::= clk_signal_name = '0' | clk_signal_name = '1'
 event_expr  ::= clk_signal_name'EVENT | not clk_signal_name'STABLE
```

The RISING_EDGE and FALLING_EDGE functions are as declared by the package STD_LOGIC_1164 of IEEE Std 1164-1993.

6.1.2.1 Rising (positive) edge clock

The following expressions shall represent a rising edge clock:

a) RISING_EDGE(*clk_signal*_name)

b) *clk_signal*_name = '1' **and** *clk_signal*_name'EVENT

c) *clk_signal*_name'EVENT **and** *clk_signal*_name = '1'

d) *clk_signal*_name = '1' **and not** *clk_signal*_name'STABLE

e) **not** *clk_signal*_name'STABLE **and** *clk_signal*_name = '1'

6.1.2.2 Falling (negative) edge clock

The following expressions shall represent a falling edge clock:

a) FALLING_EDGE(*clk_signal*_name)

b) *clk_signal*_name = '0' **and** *clk_signal*_name'EVENT

c) *clk_signal*_name'EVENT **and** *clk_signal*_name = '0'

d) *clk_signal*_name = '0' **and not** *clk_signal*_name'STABLE

e) **not** *clk_signal*_name'STABLE **and** *clk_signal*_name = '0'

6.1.3 Modeling edge-sensitive storage elements

An edge-sensitive storage element may be modeled either by a signal or variable that is updated at a clock edge.

Definitions:

<**sync_condition**>. A <boolean_expression> with a <clock_edge> expression that only is TRUE when <clock_edge> is TRUE.

<**async_condition**>. A <boolean_expression> without a <clock_edge> expression.

<**sync_assignment**>. An assignment to a signal or variable that is controlled explicitly by <clock_edge> in all execution paths.

<**async_assignment**>. An assignment to a signal or variable that is not controlled by <clock_edge> in any execution path.

To illustrate these definitions, here are two examples:

Example of <async_assignment>:

```
SimpleEdgeModel: process: (clk, reset)
                   begin
                   if( rising_edge(clk) and reset = '0' ) then
                      Q <= D;                      -- sync assignment
                   elsif( reset = '1' ) then
                      Q <= '0';                    -- async assignment
                   end if;
                 end process;
```

In this example, the assignment `Q <= '0'` is controlled by `reset = '1'` but not by the <clock_edge> as represented by `rising_edge(clk)`. Notice that when `reset is '1'`, `rising_edge(clk)` may be TRUE or FALSE; therefore, the assignment in the **elsif** is asynchronous.

Example of <async_condition>:

```
ComplexEdgeModel:
   process( clk, en, reset )
      begin
      if (en = '1' and rising_edge(clk))
                       or (en = '1' and reset = '1')  then
         if (reset = '1') then
            Q <= '0';                             -- async assignment
         elsif (en = '1' and rising_edge(clk)) then -- sync condition
            Q <= D;                               -- sync assignment
         end if;
      end if ;
   end process ;
```

In this example, the <sync_condition> is the boolean expression "`en = '1' and rising_edge(clk)`", because it can be true only when the clock edge also is true. The <async_condition> is the boolean expression "`en = '1' and reset = '1'`" **and**ed with `reset = '1'`". With these controlling the execution flow, the assignment, "`Q <= '0'`" is an <async_assignment> because it is executed when the <async_condition> is true, and the assignment "`Q <= D`" is a <sync_assignment> because it is executed when the <sync_condition> is true.

NOTE—An edge-sensitive storage element inferred for a variable may be eliminated during optimization if there exists another edge-sensitive storage element with its same functionality.

6.1.3.1 Edge-sensitive storage from a process with sensitivity list and one clock

Edge-sensitive storage shall be modeled for a signal or variable assigned inside a process with sensitivity list when all of the following apply:

a) The signal or variable has a <sync_assignment>.

b) There is no execution path in which the value update from a <sync_assignment> overrides the value update from an <async_assignment> unless the <async_assignment> is an assignment to itself.

c) It is possible to statically enumerate all execution paths to the signal or variable assignments.

d) The process sensitivity list includes the clock and any signal controlling an <async_assignment>.

e) The <clock_edge> is present in the conditions only, and the <clock_edge> always expresses the same edge of the same clock signal.

f) For a variable, the value written by a given clock edge is read during a subsequent clock edge.

NOTES

1—Except for a clock signal, signals read in a <sync_assignment> or signals controlling a <sync_assignment> are not required to be on the process sensitivity list.

2—In rule b) above, an <async_assignment> of a signal to itself is an exception because self-assignment retains the previous value, allowing a future, newly clocked <sync_assignment> value to replace a definite previous value. This specific kind of <async_assignment> thus merely continues the storage state previously established; it has no effect on any stored value, so overriding it makes no difference.

3—The <clock_edge> may be in a sequential procedure.

Example 1: Storage may be assigned in multiple statements in a process.

```
TwoReg : process(clk)
  begin
  if rising_edge(clk) then
    Q1 <= D1;
    Q2 <= D2;
  end if;
end process;
```

Example 2: Multiple statements in a process, with a reset.

```
TwoRegReset : process(clk, reset)
  begin
  if rising_edge(clk) then
    Q1 <= D1;
    Q2 <= D2;
  end if;

  if reset = '1' then
    Q1 <= '0';
  end if;
end process;
```

Example 3: A signal (or variable) may be updated with multiple <clock_edge> conditions on the same edge of the clock.

```
EnableEdgeProc : process(clk, reset)
  begin
  if reset = '1'
    then Q <= '0';
    else
    case sel is
    when '0' => if rising_edge(clk) then Q <= D0; end if;
    when '1' => if rising_edge(clk) then Q <= D1; end if;
    when others => Q <= '0';
    end case;
  end if;
end process;
```

Example 4: More complicated multiple <clock_edge> conditions.

```
-- clk   reset  e1    e2  ||     Q
--
-- *     1      *     *   ||     0
-- rise  0      1     *   ||    D11
-- !rise 0      1     *   ||    hold  ## <clock_edge> OK as per rule b.
-- rise  0      0     1   ||    D12
-- !rise 0      0     1   ||    hold  ## <clock_edge> OK as per rule b.

multiEnableEdgeProc : process(clk, reset)
  begin
  if reset = '1' then
    Q <= '0';
  elsif e1 = '1' and rising_edge(clk) then
```



```
      Q <= D11;
    elsif e2 = '1' and rising_edge(clk) then
      Q <= D12;
    end if;
  end process;
```

Example 5: Async and sync assignments controlled by complicated boolean expressions.

```
RegProc5 : process( clk, reset )
begin
  if ( (en = '1' and rising_edge(clk)) or reset = '1') then
   if ( reset = '1' ) then
      Q <= '0'; -- async assignment.
   elsif (en = '1' and rising_edge(clk)) then -- sync condition
      Q <= D; -- sync assignment
   end if;
  end if ;
end process ;
```

Incorrect Example 6: Violates rule a). Is not a <sync_assignment> because it is not controlled by a <clock_edge> in all execution paths.

```
IllegalRegProc6 : process( clk, reset )
begin
  if ( rising_edge(clk) or reset = '1') then
   if ( reset = '1' ) then
      Q <= '0';
   else
      Q <= D;
   end if;
  end if ;
end process ;
```

Example 7: Sequential statements are allowed in a process outside the statement defining the edge-sensitive storage element(s).

```
ComboResetDFF:
     process (clock, reset1, reset2, set, async_preload, A, Q)
        variable RESET : std_logic;
        begin
        RESET := reset1 or reset2;       -- Outside the edge statement
        if RESET = '1' then
              Q <= '0';
           elsif set = '1' then
              Q <= '1';
           elsif async_preload = '1' then
              Q <= A;
           elsif rising_edge(clock) then
              Q <= D;
        end if;
        QBAR <= not Q;                   -- Outside the edge statement
        end process;
```

6.1.3.2 Edge-sensitive storage using a single wait statement

Assume the wait statement to be one of the following:

a) Wait statement with explicit clock edge:

```
1) wait until [ <async_condition> or ] <sync_condition>;
2) wait  on <async_signals>, <clock_signal> [, <sync_signals>]
         until <async_condition> or <sync_condition>;
3) wait  on  <clock_signal> [, <sync_signals>]
         until  <sync_condition>;
```

where

```
<async_condition> is defined in 6.1.3.
<sync_condition>  is defined in 6.1.3.
<async_signals>  is a sensitivity list with signals present
                   in the <async_condition>.
<clock_signal>  is a sensitivity list with clock signal present
                   in the <clock_edge>.
<sync_signals>  is a sensitivity list with signals present
                   in the <sync_condition> excluding the <clock_signal>.
<clock_edge>  is defined in 6.1.2.
```

b) Wait statement with implicit clock edge:

```
1) wait until <clock_level> ;
2) wait on <clock_level_signal>  until <clock_level_expr>;
```

where

```
<clock_level_signal> ::= sensitivity list with clock signal present
                          in the <clock_level>.
<clock_level>      ::=  signal_name = '0' | signal_name = '1'.
<clock_level_expr> ::= <boolean_expression> with <clock_level>, which
                        shall be TRUE only when <clock_level> is TRUE.
```

c) Wait statement without clock edge: This includes forms of wait statement from either item a) or item b) above in 6.1.3.2, in which the clock edge is not specified either explicitly or implicitly.

```
1) wait on <sensitivity_list> ;
2) wait until <condition> ;
3) wait on <sensitivity_list> until <condition> ;
```

An if statement following one of these wait statements must have <clock_edge> in the condition.

Edge-sensitive storage shall be modeled for a signal or variable, assigned inside a process with wait statement, when an assumption above is fulfilled; and, in addition:

a) The wait statement is the first or last statement of the process

b) The process with wait statement can be transformed to a process with "wait on <sensitivity_list>". The resulting process with "wait on <sensitivity_list>" must adhere to the rules in 6.1.3.1.

The transformation is described as follows:

T1. A wait statement describing an implicit clock edge model [b) above] is represented as an explicit clock edge model (A above). This can be achieved by replacing "`clk_signal_name = '0'`" with `falling_edge(clk_signal_name)`, or replacing "`clk_signal_name = '1'`" with `rising_edge(clk_signal_name)`.

T2. A wait statement of the form, "**wait until** <condition>", is transformed to an equivalent wait statement of the form, "**wait on** <sensitivity_list> **until** <condition>".

T3. After these alterations, the "**wait on** <sensitivity_list> **until** <condition>" statement is transformed to an equivalent "**wait on** <sensitivity_list>" as follows:

```
process
    <process_declarative_part>
begin
wait on <sensitivity_list> until  <condition>;
    <sequence_of_statements>;
end process;
```

transforms to this equivalent process with "**wait on** <sensitivity_list>":

```
process
   <process_declarative_part >
begin
wait on <sensitivity_list>;
if <condition> then
    <sequence_of_statements>;
end if;
end process;
```

Example: Showing the transformation to wait on

```
process
 begin
 wait on SET, reset, clock
       until SET = '1' or reset = '1' or rising_edge(clock);
 if reset = '1' then
   Q <= '0';
 elsif SET = '1' then
   Q <= '1';
 elsif  rising_edge(clock) then
   Q <= D;
 end if;
end process;
```

Using the transformations described above, the goal is the following equivalent process with only a "**wait on** <condition>" statement:

```
process
  begin
  wait on <sensitivity_list> ;
  <statement_list>
end process ;
```

This is accomplished as follows:

```
process
 begin
 wait on SET, reset, clock ;
 if SET = '1' or reset = '1' or rising_edge(clock) then
   if reset = '1' then
      Q <= '0';
   elsif SET = '1' then
      Q <= '1';
   elsif  rising_edge(clock) then
```



```
          Q <= D;
       end if;
     end if;
    end process;
```

6.1.3.3 Edge-sensitive storage with one or more clocks

Multiple if statements with different clock edge conditions may be used to update a signal or variable inside a process. The process may have a sensitivity list, or it may have an equivalent **wait on** as its first or last statement.

The clock edge conditions shall be mutually exclusive.

For each clock edge expression, when the remaining clock edge expressions are replaced by FALSE in all statements of the process, the transformed process must fulfill the conditions of 6.1.3.1 or 6.1.3.2.

The signal in the first clock edge expression (textually) shall be taken as the functional clock.

NOTES

1—It is recommended to have simulation specific code enclosed within RTL_SYNTHESIS OFF/ON pragmas to check the mutual exclusivity.

2—The determination of the functional clock is made on a process-by-process basis; the intended functional clock has to be coded first in each process.

Example 1: Two different clock signals

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity multi_clock_ff_example is
  port(reset, clk1, clk2,
       data1, data2 : in  Std_Logic;
                  Q : out Std_Logic );
  end;

architecture RTL of multi_clock_ff_example is
  begin
  -- Process sensitive to controlling signals reset, clk1 and clk2
  process( reset, clk1, clk2 )
    begin
      if reset = '1' then
         Q <= '0' ;
      elsif rising_edge(clk1) then
         Q <= data1 ;
      elsif rising_edge(clk2) then
         Q <= data2 ;
      end if ;

      -- RTL_SYNTHESIS OFF
      if  rising_edge(clk1) and rising_edge(clk2) then
        assert (TRUE) report
     "Warning: Scan and functional clock are active together"
        severity Warning ;
        Q <= 'X' ;
```



```
        end if ;
        -- RTL_SYNTHESIS ON

      end process;
    end RTL;
```

Example 2: Two different edges of one clock signal

```
DualEdge_Proc: process (Clk, Reset) is
  begin
    if Reset = '1' then
          Q <= (others => '0');
    elsif rising_edge(Clk) then
          Q <= D4Rise;
    elsif falling_edge(Clk) then
          Q <= D4Fall;
    end if;
  end process DualEdge_Proc;
```

6.1.3.4 Edge-sensitive storage with multiple waits

When modeling edge-sensitive storage elements using multiple wait statements, the following rules shall apply:

a) The wait statement shall be modeled according to 6.1.3.2, item a) or item b).

NOTE–The **wait** may reside in a sequential procedure.

b) If one wait statement uses an <async_condition>, all wait statements shall use the same, identical <async_condition>.

c) Each wait statement shall specify the same clock edge of a single clock.

d) Statements under each wait statement to handle asynchronous condition (i.e., signals from <async_condition>) shall be the same.

NOTE–An **exit** or **next** following each **wait** may be used to implement a full reset of a state machine.

e) If simulation semantics require that the value of a variable being read is written on the previous <clock_edge>, edge-sensitive storage shall be modeled for it.

Example 1: A multicycle data path element:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

Entity Mult is
   port (
           clk : in  std_logic;
         start : in  std_logic;
          done : out std_logic;
          A, B : in  unsigned (3 downto 0);
             Y : out unsigned (7 downto 0)
        );
end Mult;
--
```

```
Architecture ImplicitFSM of Mult is
   signal intY : unsigned(7 downto 0);
   begin
   MultProc : process
      begin
      wait until clk = '1';
      if start = '1' then
        done <= '0';
        intY <= (others => '0');
        for i in A'range
          loop
          wait until clk = '1';
          if A(i) = '1' then
            intY <= (intY(6 downto 0) & '0') + B ;
          else
            intY <= (intY(6 downto 0) & '0') ;
          end if;
          end loop;
        done  <= '1';
        end if;
      end process;
   Y <= intY  ;      -- final state Y = A * B
   end ;
```

Example 2: Asynchronous reset modeling.

```
generericStateMachineProc: process
  begin
    RESET_LOOP: loop
        if reset = '1' then  -- reset/init  state
          Y <= '0';
          X <= '0';
        end if;

        wait until reset = '1' or rising_edge(clk);
        next RESET_LOOP when ( reset = '1' );
        X <=  A; -- state one

        wait until reset = '1' or rising_edge(clk);
        next RESET_LOOP when ( reset = '1' );

        Y <= B;  -- state two
        wait until reset = '1' or rising_edge(clk);
    end loop RESET_LOOP;
 end process;
```

Example 3: Serial transmission modeling.

```
------------------------------------------------------------
  UartTxFunction : Process
------------------------------------------------------------
  begin
    TopLoop : loop
      if (nReset = '0') then
        SerialDataOut <= '1' ;
        TxRdyReg      <= '1' ;
      end if ;

      wait until nReset = '0' or
                 (rising_edge(UartTxClk) and DataRdy = '1') ;
      next TopLoop when nReset = '0' ;
      SerialDataOut  <= '0';
      TxRdyReg       <= '0';

      -- Send 8 Data Bits
      for i in 0 to 7 loop
        wait until nReset = '0' or rising_edge(UartTxClk) ;
        next TopLoop when nReset = '0';
        SerialDataOut  <= DataReg(i) ;
        TxRdyReg       <= '0' ;
      end loop ;

      -- Send Parity Bit
      wait until nReset = '0' or rising_edge(UartTxClk) ;
      next TopLoop when nReset = '0' ;
      SerialDataOut  <=
        DataReg(0) xor DataReg(1) xor DataReg(2) xor
        DataReg(3) xor DataReg(4) xor DataReg(5) xor
                       DataReg(6) xor DataReg(7) ;
      TxRdyReg       <= '0';

      -- Send Stop Bit
      wait until nReset = '0' or rising_edge(UartTxClk) ;
      next TopLoop when nReset = '0';
      SerialDataOut  <= '1' ;
      TxRdyReg       <= '1' ;

    end loop ;
  end process ;
```

6.1.3.5 Edge-sensitive storage using concurrent signal assignment statements

A concurrent conditional signal assignment statement may be used to model an edge-sensitive storage element provided that the assignment can be mapped to a process that adheres to the rules in 6.1.3.1.

Example:

```
COND_SIG_ASSGN: Q <= '0' when RESET = '1'       else
                     '1' when SET = '1'         else
                      A  when ASYNC_LOAD = '1' else
                      D  when CLOCK'EVENT and CLOCK = '1';
```

6.1.3.6 Edge-sensitive storage using a guarded block

A signal assigned in a guarded block shall model edge-sensitive storage if the equivalent process in the block fulfills the rules in 6.1.3.1 and the target signal is declared of kind **register**. The guard expression must be in the following form:

```
<guard_exp> ::= [<async_condition> or]<guard_sync_condition>
<guard_clk_edge> ::= not <clock_signal>'stable and <clock_signal> = '0'
                   | not <clock_signal>'stable and <clock_signal> = '1'

<guard_sync_condition> ::= A <boolean_expression> which includes
                         <guard_clk_edge> expression and which is TRUE
                         only when <guard_clk_edge> is TRUE.
<async_condition> ::= as defined in 6.1.3.
```

Example:

```
architecture GUARD1 of top is
    signal Q : std_logic register;
    begin
    guardedRegBlock:
      block( set = '1' or reset = '1' or not clk'stable and clk = '1' )
        begin
        Q <= guarded '1' when   set = '1' else
                     '0' when reset = '1' else
                      D ;
        end
      block;
    end;
```

6.1.3.7 Edge-sensitive storage from a concurrent subprogram

Edge-sensitive storage shall be modeled for a signal assigned in a concurrent procedure call that can be mapped to a process adhering to the rules in 6.1.3.1.

NOTES

1—A wait in a concurrent subprogram should be used with care: Both the concurrent statement and the wait statement have sensitivity lists.

2—Recursive subprograms are supported if and only if the subprogram can be statically inlined, as required in 8.2.2.

Example:

```
architecture CONCUR_SUB of flipflop is
procedure FF
  (signal   clk,
          reset,
              D : in  std_logic;
   signal     Q : out std_logic
  ) is
begin
  if reset = '1' then
        Q <= '0';
  elsif rising_edge(clk) then
        Q <= D;
  end if;
end FF;

signal reg1, reg2, reg3 : std_logic;

begin

 FF( clk, reset,  D,   reg1 );
 FF( clk, reset, reg1, reg2 );
 FF( clk, reset, reg2, reg3 );
 FF( clk, reset, reg3,   Q  );

end;
```

6.2 Level-sensitive sequential logic

6.2.1 Modeling level-sensitive storage elements

6.2.1.1 Level-sensitive storage from process with sensitivity list

A level-sensitive storage element shall be modeled for a signal (or variable) when all the following apply:

a) The signal (or variable) has an explicit assignment.

b) The signal (or variable) does not have an execution path with <clock_edge> as a condition.

c) There are executions of the process that do not execute an explicit assignment (via an assignment statement) to the signal (or variable).

By default, the effect of an identity assignment of the signal (or variable) shall be as though the assignment was not present.

If the combinational attribute decorates the signal (or variable), combinational logic with feedback shall be synthesized.

The process sensitivity list shall contain all signals read within the process statement.

NOTES

1—Variables declared in subprograms never model level-sensitive storage elements because variables declared in subprograms are always initialized in every call.

2—When a signal is assigned from within a procedure it shall have the same inference semantics as a signal assignment from within a process.

3—Recursive procedure calls are allowed if and only if the procedure can be statically inlined, as required in 8.2.2.

4—It is recommended to avoid a modeling style in which the value of a signal or variable is read before its assignment. This recommendation is meant to avoid the generation of unwanted storage elements.

Example 1:

```
LEV_SENS_1: process (RESET, ENABLE, D)
            begin
              if RESET = '1' then
                Q <= '0';
              elsif ENABLE = '1' then
                Q <= D; -- Q is an incomplete asynchronous
              end if;   -- assignment, so it models a level-sensitive
                        -- storage element.
            end process;
```

Example 2:

```
-- If attribute 'combinational' is FALSE on a process,
-- identity assignment Q <= Q; causes synthesis of a latch.
-- In this example, the value TRUE causes synthesis of combinational
-- logic with feedback:

use ieee.rtl_attributes.all;            -- declaration of combinational
attribute combinational of LEV_SENS_2:label is TRUE;

LEV_SENS_2 : process ( ENABLE, D)
             begin
               if ENABLE = '1' then
                  Q <= D;
               else
                  Q <= Q ;-- identity assignment. Same as Q <= unaffected;
               end if;
             end process;
```

Example 3:

```
-- A process modeling both latch and flip-flop is supported.
RegPlusLatProc: process(clk, reset, gEnable)
  variable gLatch : std_logic;
  begin
    if clk = '0' then
      gLatch := gEnable;
    end if;
    if reset = '1' then
      Q <= '0';
    elsif gLatch = '1' and rising_edge(clk) then
      Q <= D;
    end if;
  end process;
```


Example 4:

```
-- Again, if attribute 'combinational' is FALSE on a process,
-- identity assignment 'Q_TEMP := Q_TEMP;' is replaced by null statement.

     use ieee.rtl_attributes.all;
     attribute combinational of LEV_SENS_4:label is FALSE ;

LEV_SENS_4 : process ( ENABLE , D)
               variable Q_TEMP : BIT ;
               begin
               if ENABLE = '1' then
                 Q_TEMP := D;
               else
                 Q_TEMP := Q_TEMP ; -- identity assignment
               end if;
               Q <= Q_TEMP ;
             end process;
```

Example 5: Inferred latch, perhaps as part of a scan chain

```
     -- Signal Q has an initial value in its declaration
     LEV_SENS_5 : process (enable, Q)
     begin
       if enable = '1' then
         Q <= Q ;
       end if;
     end process;
```

Example 6:

```
     -- Ram element as level-sensitive storage is supported.

       RAM_WRITE : process (WDE,D,ADDR)
                   begin
                      if WDE = '1' then
                        myMem( ADDR ) <= D;
                      end if;
                   end process;
                   Q <= myMem(ADDR);
```

6.2.1.2 Level-sensitive storage from concurrent signal assignment

A level-sensitive storage element shall be modeled for a signal that is assigned in a concurrent signal assignment statement that can be mapped to a process that adheres to the rules in 6.2.1.1.

Example 1:

```
     LEV_SENS_7:  Q <= '0' when RESET='1' else -- This is identical
                        D  when ENABLE;        -- to LEV_SENS_1 in 6.2.1.1,
                                                  above.
```

Example 2:

```
     LEV_SENS_8:  With ENABLE select
                     Q <=  D when '1',
                           Q when others; --Identical to LEV_SENS_2 in  6.2.1.1,
                                          -- and models combinational logic.
```

Example 3:

```
LEV_SENS_9:  with ENABLE select
                  Q <=    D        when '1',
                      unaffected   when others;
```

6.2.1.3 Level-sensitive storage from concurrent procedure call

A level-sensitive storage element shall be modeled for a signal assigned in a concurrent procedure call that can be mapped to a process that adheres to the rules in 6.2.1.1.

Example:

```
architecture CONCUR_SUB of level_sensitive
  is
  procedure latch(signal ENABLE, D :in  bit;
                  signal          Q :out bit ) is
     begin
     if ENABLE ='1' then Q <= D;
     end if;
     end;
  signal Q : bit;
  begin
    latch(a,sel,Q); --Concurrent procedure call which models a latch.
  end CONCUR_SUB;
```

6.2.1.4 Level-sensitive storage from guarded block

A level-sensitive storage element shall be modeled for a signal assigned in a guarded block that can be mapped to a process fulfilling the rules in 6.2.1.1. The signal must be of kind **register** and a <clock_edge> expression must not be used in the guard expression.

Each concurrent guarded signal assignment statement within such a guarded block must be equivalent to a process statement fulfilling the rules in 6.2.1.1.

Example:

```
entity latchEntity is
  port ( clk1, dat1,
               dat2 : in  Std_Logic;
             q1, q2 : out Std_Logic );
  end
entity latchEntity;

architecture latchArch of latchEntity is
  signal gq1, gq2 : Std_Logic register;
  begin
  d1Latch:
    block(clk1 = '1')
    begin
    gq1 <= guarded dat1;
    gq2 <= guarded dat1 and not dat2;
    end block
```



```
d1Latch;
q1 <= gq1;
q2 <= gq2;
end latchArch;
```

6.3 Three-state logic and busses

6.3.1 Three-state logic from 'Z' assignment

Three-state logic shall be modeled when an object or an element of the object is explicitly assigned the IEEE Std 1164-1993 value 'Z'. The target signal shall be of type Std_Logic.

The assignment to 'Z' shall be a conditional assignment.

For a signal that has multiple drivers, if one driver has an assignment to 'Z', every driver of that signal shall be assigned a 'Z' under at least one condition.

NOTE—If an object is assigned a value 'Z' in a process that is edge-sensitive or level-sensitive, as described in 6.1 and 6.2, a synthesis tool may infer sequential elements on all inputs of the three-state logic.

6.3.2 Three-state logic from guard disconnect

Three-state logic may be modeled by a guarded signal assignment to a target signal of kind **bus**. When the guard condition is false, the driver is removed (disconnected), which is equivalent to a high-impedance value. The target signal shall be of type Std_Logic.

It shall be an error if any target signal of a guarded assignment is not declared explicitly of kind **bus** or **register**. When declared of kind **register**, it shall be an error if the rules for sequential logic in 6.1.3.6 or 6.2.1.4 are not fulfilled.

Example:

```
signal dout : Std_Logic bus; -- Kind "bus" yields three-state;
                             -- kind "register" yields sequential logic
signal flag : Std_Logic;     -- (see 6.2.1.4).
tri_state: block (en = '1')
           begin
             dout <= guarded din;
             flag <= en;
           end block;
```

6.4 Combinational logic

Any process that does not contain a clock edge or wait statement shall model either combinational logic or level-sensitive sequential logic.

If there is always an assignment to a variable or signal in all possible executions of the process and all variables and signals have well-defined values, then the variable or signal models combinational logic.

a) If the variable or signal is updated before it is read in all executions of a process, then it shall model combinational logic.
b) If a variable or signal is read before it is updated, then it may model combinational logic.

For combinational logic, the process sensitivity list shall list all signals read within the process statement.

6.5 ROM and RAM memories

6.5.1 Read-only memory (ROM)

An asynchronous ROM shall be modeled using one of the following styles:

a) **Constant** declaration of a memory array. A ROM instance may be generated when the memory array is read from within a concurrent statement.

b) One-dimensional array with data in **case** statement.

The *rom_block* attribute shall be used to identify the variable that models the ROM. See 7.1.5.1 for this attribute. If the *logic_block* attribute is used, then it shall imply that no ROM is to be inferred.

NOTES

1—The standard does not define how or in what form the ROM values are to be saved after synthesis when the *rom_block* attribute is used.

2—In the absence of a *rom_block* or *logic_block* attribute, there is no constraint on the synthesized ROM implementation.

6.5.1.1 ROM with constant array

The values of the ROM may be defined within a constant defined as an array of arrays, or as an array of integers or bits.

Example:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ROMconst is
  port (
  Z : out std_logic_vector(3 downto 0);
  A : in std_logic_vector(2 downto 0));
end entity ROMconst;

architecture RTL of ROMconst is
  type mem_typ is array(0 to 7) of std_logic_vector(3 downto 0);
  constant ROMINIT : mem_typ :=
    (  0   => "1011",
       1   => "0001",
       2   => "0011",
       3   => "0010",
       4   => "1110",
    others => "0000");
  attribute rom_block: string; --OK not to use ieee.rtl_attributes package,
                               --but the attribute must be defined
                               --identically as in the package.
  attribute rom_block of ROMINIT : constant is "ROM_CELL_XYZ01";
  -- For ROM design with combinational logic use:
  -- attribute logic_block of ROMINIT : constant is TRUE;
```

```
    begin
    Z <= ROMINIT(TO_INTEGER(UNSIGNED(A)));
    end
  architecture RTL;
```

6.5.1.2 ROM with case statement

In this style, the data values of a ROM shall be defined within a case statement. All the values of the ROM shall be defined within the case statement. The value assigned to each ROM address shall be a static expression. The object (signal or variable) attributed with the *rom_block* attribute models the ROM. The address of the ROM shall be the same as the **case** expression. The ROM variable is the data. The case statement may contain other assignments or statements that may or may not affect the ROM variable. However, all assignments to the ROM object shall be done within only one case statement.

Example 1: ROM defined by a signal:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.rtl_attributes.all; -- For declaration of rom_block attribute.

entity ROM is
  port (
    Z : out std_logic_vector(3 downto 0);
    A : in  std_logic_vector(2 downto 0));
end entity ROM;

architecture RTL of ROM is
  attribute rom_block of Z : signal is "ROM32Kx16";
  -- For ROM design with combinational logic use:
  -- attribute logic_block of Z : signal is TRUE;
  begin -- architecture RTL
  Rom_Proc : process (A) is
    begin -- process Rom_Proc
    case A is
      when "000" => Z <= "1011";
      when "001" => Z <= "0001";
      when "100" => Z <= "0011";
      when "110" => Z <= "0010";
      when "111" => Z <= "1110";
      when others => Z <= "0000";
    end case;
  end process Rom_Proc;
end architecture RTL;
```

Example 2: ROM defined by a variable:

```
architecture RTL of ROM is
  begin -- architecture RTL
  Rom_Proc : process (a)
    is
    variable rom : std_logic_vector(3 downto 0);
    attribute rom_block        : string;  -- OK not to use rtl_attributes.
    attribute rom_block of rom : variable is "ROM_CELL_XYZ01";
    -- For ROM design with combinational logic use:
    -- attribute logic_block of rom : variable is TRUE;
    begin -- process Rom_Proc
    case a is
      when "000" => rom := "1011";
      when "001" => rom := "0001";
      when "100" => rom := "0011";
      when "110" => rom := "0010";
      when "111" => rom := "1110";
      when others => rom := "0000";
    end case;
    Z <= rom;
  end process Rom_Proc;
end architecture RTL;
```

NOTE—See 7.1 for additional information on the definition of the synthesis attributes.

6.5.2 Random-access memory (RAM)

A RAM shall be modeled using a signal or a variable that may have the attribute *ram_block* associated with it. See 7.1.5.2 for this attribute. The values of the RAM may be defined within an array of arrays, or as an array of integers or bits. A RAM element may either be modeled as an edge-sensitive storage element or as a level-sensitive storage element. A RAM data value may be read synchronously or asynchronously.

NOTES

1—An attribute may be necessary to identify the RAM style. If combinational logic is desired instead of a RAM, use the attribute *logic_block* instead of the attribute *ram_block*.

2—In the absence of a *ram_block* or *logic_block* attribute, there is no constraint on the synthesized implementation.

Example 1: A RAM with edge-sensitive write to storage elements

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.rtl_attributes.all;

entity ram is
  generic (
    WIDTH : Natural :=  8;
    DEPTH : Natural := 16);
  port (
     q : out std_logic_vector(WIDTH-1 downto 0); -- Ram output
```



```
    d : in  std_logic_vector(WIDTH-1 downto 0); -- Ram data input
    a : in  std_logic_vector(DEPTH-1 downto 0); -- Address
   we : in  std_logic;                          -- Write enable
  clk : in  std_logic);                         -- system clock
end entity ram;

architecture RTL of ram is
  type ram_typ is array(0 to 2**DEPTH-1) of
                        std_logic_vector(WIDTH-1 downto 0);
  signal ram : ram_typ; -- ram element
  attribute ram_block of ram : signal is "RAM_CELL XYZ01";
  -- For RAM design with registers logic use:
  -- attribute logic_block of z : signal is TRUE;
  begin -- architecture RTL
        -- purpose: Synchronous Ram definition
        -- type : combinational
  Ram_Proc: process is
    begin -- process Ram_Proc
    wait until clk = '1';
    if we = '1' then
      ram(to_integer(unsigned(a))) <= d;
    end if;
  end process Ram_Proc;
  q <= ram(to_integer(unsigned(a)));
end architecture RTL;
```

Example 2: A RAM with edge-sensitive read and write storage elements

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.rtl_attributes.all;
entity ram is
  generic (
    WIDTH : Natural :=  8;
    DEPTH : Natural := 16);
  port (
    q : out std_logic_vector(WIDTH-1 downto 0); -- Ram output
    d : in  std_logic_vector(WIDTH-1 downto 0); -- Ram data input
    a : in  std_logic_vector(DEPTH-1 downto 0); -- Address
   we : in  std_logic;                          -- Write enable
   re : in  std_logic;                          -- Read enable
  clk : in  std_logic);                         -- system clock
end entity ram;
--
architecture RTL of ram is
  type ram_typ is array(0 to 2**DEPTH-1) of
                        std_logic_vector(WIDTH-1 downto 0);
  constant Zvec : std_logic_vector(WIDTH-1 downto 0) := (others=>'Z');
  constant Xvec : std_logic_vector(WIDTH-1 downto 0) := (others=>'X');
  begin -- architecture RTL
        -- purpose: Synchronous Ram definition
  Ram_Proc: process is
    variable ram   : ram_typ; -- ram element
```

```
    variable q_int : std_logic_vector(WIDTH-1 downto 0);
    attribute ram_block of ram : variable is "RAM_CELL XYZ01";
    -- For RAM design with register logic use
    -- attribute logic_block of ram : variable is TRUE;
    begin -- process Ram_Proc
    wait until clk = '1';
    q_int <= ram(to_integer(unsigned(a)));
    if we = '1' then
      ram(to_integer(unsigned(a))) <= d;
    end if;
  end process Ram_Proc;
  --
  q <= q_int when '1'
       Zvec  when '0'
       Xvec  when others;
end architecture RTL;
```

Example 3: A RAM with level-sensitive storage elements:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.rtl_attributes.all;

entity ramlatch is
  generic (
          WIDTH : Natural := 8;
          DEPTH : Natural := 16);
  port (
        q : out std_logic_vector(WIDTH-1 downto 0); -- Ram output
        d : in  std_logic_vector(WIDTH-1 downto 0); -- Ram data input
        a : in  std_logic_vector(DEPTH-1 downto 0); -- Address
       we : in  std_logic -- Write enable
       );
end entity ramlatch;

architecture RTL of ramlatch is
  type ram_typ is array(0 to 2**DEPTH-1) of
                             std_logic_vector(WIDTH-1 downto 0);
  signal ram : ram_typ; -- ram element
  attribute ram_block of ram : signal is ""; -- tech mapper decides
  -- For RAM design with register logic use:
  -- attribute logic_block of ram : signal is TRUE;
  begin -- architecture RTL
        -- purpose: Asynchronous Ram definition
  Ram_Proc: process (a, d, we) is
  begin -- process Ram_Proc
    if we = '1' then
      ram(to_integer(unsigned(a))) <= d;
```



```
        end if;
      end process Ram_Proc;
      q <= ram(to_integer(unsigned(a)));
    end architecture RTL;
```

7. Pragmas

Pragmas commonly are used to aid the synthesis tool in interpreting and implementing the VHDL model of a design. Pragmas can take the form of attributes or metacomments.

7.1 Attributes

For the boolean-valued attributes described in this subclause, the effects refer to a value of TRUE; a boolean FALSE value of the attribute shall result in behavior identical to the behavior when the attribute specification has been omitted entirely.

User-defined attributes shall be ignored, except the synthesis-specific attributes in this subclause. All declarations of the synthesis-specific attributes have been collected in Annex B. Any declaration in Annex B may be located anywhere the user decides is appropriate; however, declarations copied from Annex B shall be identical to those in that Annex.

7.1.1 Hierarchy control attributes

7.1.1.1 Keep attribute

Attribute name: KEEP

Attribute subtype: boolean

Decorated item: entity, component declaration, component instantiation, signal, variable

The KEEP attribute shall indicate to the synthesis tool that the decorated item shall be preserved, and not deleted or replicated. When decorating an entity, component declaration, or component instantiation, the internals of the decorated item shall not be subject to optimization. This attribute may be used to decorate portions of the design that have been previously synthesized and are being reused in the current design.

When this attribute is found decorating an entity, a synthesis tool shall not alter the logic in any instance of that entity. When this attribute is found decorating a component declaration, a synthesis tool shall not alter the logic of any instance of that component. When this attribute is found decorating a component instance, a synthesis tool shall not alter the logic for that instance.

7.1.1.2 Hierarchy creation attribute

Attribute name: CREATE_HIERARCHY

Attribute subtype: boolean

Decorated item: entity, block, subprogram, process

The attribute CREATE_HIERARCHY shall be used to indicate that the boundary around the decorated item shall be maintained. An extra level of hierarchy may be created around the logic synthesized for the decorated item; this level shall not be dissolved into that of the parent item.

7.1.1.3 Hierarchy dissolution attribute

Attribute name: DISSOLVE_HIERARCHY

Attribute subtype: boolean

Decorated item: entity, component declaration, component instantiation

The DISSOLVE_HIERARCHY attribute shall indicate to the synthesis tool that the design entity corresponding to the item decorated by the attribute should be deleted and its logic instantiated in the parent of the decorated item. This attribute can used to denote portions of the design that would better serve the design goals by being dissolved into a higher hierarchical level.

When this attribute is used to decorate an entity, all instances of that entity shall be dissolved at the next higher hierarchical level. When this attribute is used to decorate a component declaration, all instances of that component shall be dissolved in their respective immediately enclosing design units. When this attribute is used to decorate a component instance, only the named entity bound to that instance shall be dissolved.

NOTE—A hierarchy control attribute may not have any effect if by default a synthesis tool exhibits the attribute's behavior.

7.1.2 Register implementation attributes

7.1.2.1 Interconnection attributes

Definitions for the purpose of this subclause:

set logic: the logic that sets the output of a storage device to 1.

reset logic: the logic that sets the output of a storage device to 0.

7.1.2.1.1 For edge-sensitive storage elements

Attribute name: SYNC_SET_RESET

Attribute subtype: boolean

Decorated item: signal, process, block, entity

This attribute may be used to identify the set/reset logic of an edge-sensitive storage device so that the logic can be connected directly to the set/reset pin(s) rather than being used as an input gating condition of the edge-sensitive storage device. If the attribute is used to decorate a signal, then that signal shall be connected directly to the set/reset terminal(s) of an edge-sensitive storage device provided that a matching device is available in the synthesis library. If no matching device is available in the synthesis library, an error shall be generated.

Signal constraints: If the attribute is used to decorate a signal, and that signal does not connect to a synchronously reset or set, edge-sensitive storage device, a warning shall be generated. If the attribute is used to decorate a signal, and that signal connects to an asynchronously reset, edge-sensitive storage device, an error shall be generated. If both constraints are violated, both a warning and an error shall be generated.

If the attribute is used to decorate a process, block, or entity, the contained set/reset logic shall be connected directly to set/reset terminals of edge-sensitive storage device(s), if such devices are available in the synthesis library.

Block constraints: If the attribute is used to decorate a process, block, or entity, and that item does not imply a synchronously reset or set, edge-sensitive storage device, a warning shall be generated. If the attribute is used to decorate a process, block, or entity, and that item implies an asynchronously reset, edge-sensitive storage device, an error shall be generated. If both constraints are violated, both a warning and an error shall be generated.

NOTES

1—This attribute will not cause a synchronously reset edge-sensitve storage device to be converted to an asynchronously reset edge-senstive storage device.

2—SYNC_SET_RESET does not imply one-hot; use the ONE_HOT attribute instead. If both set and reset are present, to connect directly to the functional pins, either ONE_HOT must be specified also, or the device from the synthesis library must have identical priority to the code; otherwise, simulation mismatches may result.

Example:

```
architecture EdgeSensitive of Register is
   attribute SYNC_SET_RESET : boolean;  -- Or, use ieee.rtl_attributes.ALL.
   attribute SYNC_SET_RESET of reset : signal is true;
begin
   process(Clk)
   begin
     if rising_edge(Clk) then
       if reset = '1' then
         Q <= '1';
       else
         Q <= din;
       end if;
     end if;
   end process;
end architecture EdgeSensitive;
```

7.1.2.1.2 For level-sensitive storage elements

Attribute name: ASYNC_SET_RESET

Attribute subtype: boolean

Decorated item: signal, process, block, entity

This attribute may be used to identify the set/reset logic of a level-sensitive storage device so that the logic can be connected directly to the asynchronous set/reset pin rather than being used as an input gating condition of the level-sensitive storage device.

If the attribute is used to decorate a signal, then that signal shall be connected directly to the set/reset terminals of a level-sensitive storage device, provided that a matching device is available in the synthesis library. If no matching device is available in the synthesis library, an error shall be generated by the synthesis tool.

Signal constraints: If the decorated signal does not connect to a level-sensitive storage device, a warning shall be generated.

If the attribute is used to decorate a process, block, or entity, the described set/reset logic shall be connected directly to the set/reset terminals of level-sensitive storage device(s).

Block constraints: If the decorated item does not contain a level-sensitive storage device, a warning shall be generated. If no matching device is available in the synthesis library, an error shall be generated by the synthesis tool.

NOTE—ASYNC_SET_RESET does not imply one-hot; use the ONE_HOT attribute instead. If both set and reset are present, to connect directly to the functional pins, ONE_HOT must be specified, or the device from the synthesis library must have identical priority to the code.

Example:

```
architecture LevelSensitive of Latch is
     attribute ASYNC_SET_RESET : boolean;
     attribute ASYNC_SET_RESET of reset : signal is true;
begin
     process(reset, enable)
     begin
       if reset = '1' then
         dout <= '1';
       elsif enable = '1' then
         dout <= din;
       end if;
     end process;
end architecture LevelSensitive;
```

Without the ASYNC_SET_RESET attribute, the synthesis tool is free to produce logic as shown in Figure 1.

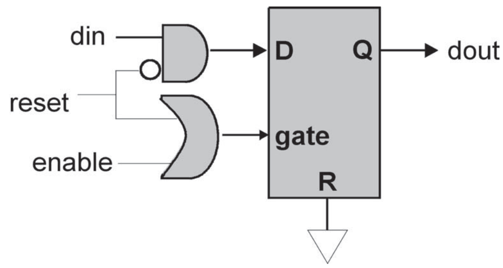

Figure 1—Latch synthesis without ASYNC_SET_RESET

Once synthesized, this result would be hard to optimize to a desirable implementation.

With the ASYNC_SET_RESET attribute, the result is as below in Figure 2. The synthesizer was not allowed to use reset to gate *enable* or *din*.

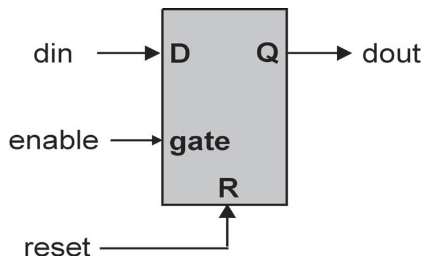

Figure 2—Latch synthesis with ASYNC_SET_RESET = TRUE

7.1.2.2 Set/reset prioritization attributes

Attribute names: ONE_HOT, ONE_COLD

*Attribute subtyp*e: boolean

Decorated item: signal

The ONE_HOT attribute identifies a collection of one-bit signals that are active high and in which only one signal in the collection is active at a given time. The ONE_COLD attribute identifies a collection of one-bit signals that are active low and in which only one signal in the collection is active at a given time.

When this attribute is used to decorate one or more signals, the synthesis tool shall not implement priority logic for these signals.

Example 1: Model of a flip-flop with *set* and *reset*.

```
-- Model with inherent priority between the set and reset:
architecture A of FLOP is
begin
  P1: process(set, reset, clock)
    begin
    if rising_edge(clk) then
      if set = '1' then  -- priority over reset
        dout <= '1';
      elsif reset = '1' then
```

```
          dout <= '0';
        else
          dout <= din;
        end if;
      end if;
    end process;
  end architecture;
```

If this priority does not match the priority of the corresponding library cell part, then the implementation will be as in Figure 3, with no attribute.

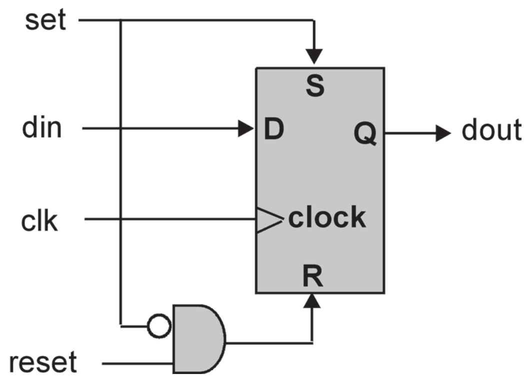

Figure 3—Flip-flop synthesis with priority mismatch and without ONE_HOT

Notice that the implementation contains logic on reset to enforce set's priority over reset.

To remove the prioritizing logic, use the ONE_HOT attribute as shown below:

Example 2: Model of the *Example 1* flip-flop without *set* priority:

```
architecture A of FLOP is
  attribute ONE_HOT : boolean;  -- Also in ieee.rtl_attributes pkg.
  attribute ONE_HOT of set, reset:  signal is true;
begin
  P1 : process(set, reset, clock)
    begin
    if rising_edge(clk) then
      if set = '1' then
        dout <= '1';
      elsif reset = '1' then
        dout <= '0';
      else
        dout <= din;
      end if;
    end if;
  end process;
end architecture;
```

The new implementation is as in Figure 4.

NOTE—This attribute also is useful in models written with asynchronous set and reset.

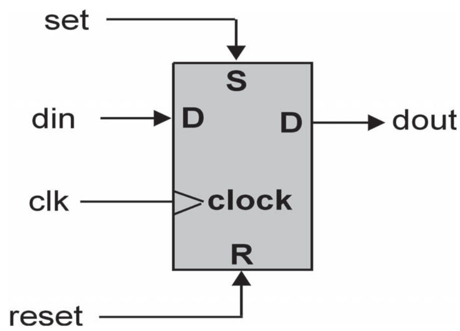

Figure 4—Flip-flop synthesis with priority mismatch and ONE_HOT

Below are some examples of the ONE_HOT attribute as used to decorate combinational logic:

Example 3: Conditional signal assignment:

```
use ieee.rtl_attributes.ALL;
entity HotEx3 is
port (
      A, B, C, D       : in  std_logic_vector(7 downto 0) ;
      S1, S2, S3, S4   : in  std_logic ;
      Y                : out std_logic_vector(7 downto 0)
     ) ;
end HotEx3;

architecture Conditional3 of HotEx3 is
   attribute ONE_HOT of S1, S2, S3, S4 : signal is true ;
begin
   Y <= A when S1 = '1' else
        B when S2 = '1' else
        C when S3 = '1' else
        D when S4 = '1' ;
end Conditional3 ;
```

Example 4: Various combinational models:

```
use ieee.rtl_attributes.ALL;
entity Combo is
port (
      A, B, C, D : in  std_logic_vector(7 downto 0) ;
      SEL        : in  std_logic_vector(3 downto 0) ;
      Y          : out std_logic_vector(7 downto 0)
     ) ;
end Combo ;

architecture Conditional4 of Combo is
   attribute ONE_HOT of SEL : signal is true ;
begin
   Y <= A when SEL(0) = '1' else
        B when SEL(1) = '1' else
        C when SEL(2) = '1' else
```

```
        D when SEL(3) = '1' ;
end Conditional4 ;

architecture Select5 of Combo is
   attribute ONE_HOT of SEL : signal is true ;
begin
   with SEL select
     Y <= A          when "0001",
          B          when "0010",
          C          when "0100",
          D          when "1000",
         "00000000" when "0000",
         "XXXXXXXX" when others ;
end Select5 ;

architecture Select6 of Combo is
   attribute ONE_HOT of SEL : signal is true ;
begin
   with SEL select
     Y <= A          when "0001",
          B          when "0010",
          C          when "0100",
          D          when "1000",
         "XXXXXXXX" when others ;
end Select6 ;
```

NOTE—Simulation mismatches may occur because of set/reset prioritization. It is the user's responsibility to ensure, maybe by writing assertions, that the behavior of the decorated signals is as anticipated.

7.1.3 Mux-selection attribute

Attribute name: INFER_MUX

Attribute subtype: boolean

Decorated item: label (of case and selected assignment statements)

A synthesis tool may determine the implementation of a case statement that assigns to the same variable or signal in all branches based on whether all possible values of the **case** expression are explicitly enumerated (usually implemented as a multiplexer) or not (usually implemented using random logic). The mux-selection attribute, when used to decorate the label of a case statement, shall direct the synthesis tool to implement the case statement with a multiplexer, regardless of the number of explicitly enumerated choices for the **case** expression.

In the example below, the signal *action* will be implemented by a multiplexer.

```
     process (STATUS)
       attribute INFER_MUX : boolean;
       attribute INFER_MUX of L1: label is true;
       begin
         L1: case  STATUS is
             when GREEN  =>  action := GO;
```



```
        when YELLOW =>  action := STEP_ON_THE_GAS;
        when others =>  action := STOP;
    end case;
  end process;
```

The following example illustrates the use of the INFER_MUX attribute with a selected assignment statement:

```
architecture A of TRAFFIC_LIGHT is
  attribute INFER_MUX : boolean;
  attribute INFER_MUX of L1: label is true;
  begin
    L1: with STATUS select
         action <= GO          when GREEN,
                   STEP_ON_GAS when YELLOW,
                   STOP        when others;
  end architecture;
```

7.1.4 Subprogram implementation attributes

Attribute name: IMPLEMENTATION, RETURN_PORT_NAME

Attribute subtype: string

Decorated item: procedure, function, label (of signal or variable assignment)

The IMPLEMENTATION attribute, when used to decorate a subprogram, shall indicate to the synthesis tool that the subprogram is to be implemented with the entity or synthesis library cell specified in the value of the attribute. The IMPLEMENTATION attribute allows the synthesis tool to ignore the body of the subprogram entirely and insert the appropriate entity or technology cell in place of the subprogram call.

Consider the following example:

```
procedure AND_OR_INVERT (A,B,C: in bit; signal O: out bit) is
  begin
    O <=  not ((A and B) or C);
  end procedure;
  . . .
AND_OR_INVERT(X, Y, Z, Q);   -- procedure call
```

The logic for the procedure call would typically be implemented with a network of Boolean gates, as shown in Figure 5.

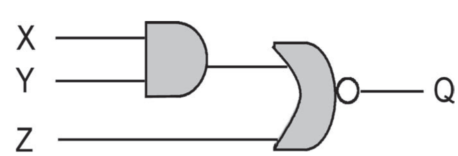

Figure 5—And-Or-Invert synthesis without IMPLEMENTATION specified

Assuming that there is an entity technology cell, AOI, that implements the same functionality, the user can direct the synthesis tool to use the same by using the IMPLEMENTATION attribute:

```
procedure AND_OR_INVERT (A,B,C: in bit; signal O: out bit) is
  begin
  O <=  not ((A and B) or C);
  end procedure;

attribute IMPLEMENTATION of
              AND_OR_INVERT: procedure is "AOI";
. . .

AND_OR_INVERT(X, Y, Z, Q);   -- procedure call
```

The synthesized design that uses the AOI cell is shown in Figure 6.

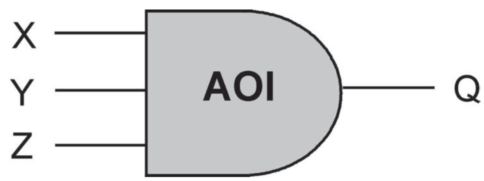

Figure 6—And-Or-Invert synthesis with IMPLEMENTATION specifying an AOI from the library

The port names and directions of the implementation cell shall be matched one-to-one to the formal parameters of the subprogram.

NOTE—It is the user's responsibility to ensure that the functionality of the subprogram is consistent with the named entity or technology cell that has been specified to implement it.

If a function is decorated with the IMPLEMENTATION attribute, the function also shall be decorated with the RETURN_PORT_NAME attribute, the value of which shall be the name of the output port of the design that communicates the return value computed by the function. In the example below, the function AND_OR_INVERT is mapped to component AOI. The RETURN_PORT_NAME attribute specifies that the value computed by the function is mapped to port O of the AOI component.

```
function AND_OR_INVERT (A,B,C: in bit) return bit is
    begin
       return  (not ((A and B) or C));
    end function;

    attribute IMPLEMENTATION of
              AND_OR_INVERT: function is "AOI";
    attribute RETURN_PORT_NAME of
              AND_OR_INVERT: function is "O";

    . . .
    Q <= AND_OR_INVERT(X, Y, Z);   -- function call
```

When the IMPLEMENTATION attribute is used to decorate the label of an assignment statement, it specifies the implementation of the operations in the right-hand side of the assignment statement. The operand designators shall match by name the ports of the specified cell.

For unary operations, the single operand shall be associated with the first port of the specified entity or synthesis library cell, whereas the result of the operation shall be associated with the second port of the specified entity or synthesis library cell.

For binary operations, the left and right operands shall be associated with the first and second ports, respectively, of the specified entity or synthesis library cell, whereas the the result of the operation shall be associated with the third port of the specified entity or synthesis library cell.

For complex expressions comprising multiple operations, the entire expression shall be implemented by the single instance of the entity or synthesis library cell specified in the attribute value. In such cases, the operands of the expression are matched to the input ports in a left to right manner.

In the example below, the right-hand side of the assignment L1 comprising two binary and operations are implemented by a single three-input technology cell “AND3”:

```
signal O, A, B, C : boolean;
attribute IMPLEMENTATION of L1 : label is "AND3";
 ...
L1 : O <= A and B and C;
```

It is an error if the specified entity or library cell is not in the target synthesis library. If the value of the IMPLEMENTATION attribute is the empty string (“”), then the attribute may be ignored by the synthesis tool. It is an error if specified entity or synthesis library cell does not have the same number of input ports as the number of operands in the right-hand side expression, or it does not have exactly one output that represents the value computed for the expression.

7.1.5 Memory modeling attributes

The attribute value of ROM_BLOCK or RAM_BLOCK may be a string name or a null string. If the string is nonnull, it shall be an error if the synthesis tool cannot find an exactly matching object in the synthesis library. An exact match in this context means that the same default binding rules specified in IEEE Std 1076-2002 for binding component and instance names is used. If the string is null, the synthesis tool shall search for a usable object (rom or ram, respectively) matching the decorated item; if no match is found, the tool shall issue a warning and then may synthesize the item as if the attribute (ROM_BLOCK or RAM_BLOCK) was not present.

It is an error if a decorated item has more than one of RAM_BLOCK, ROM_BLOCK, and LOGIC_BLOCK attributes.

7.1.5.1 ROM

Attribute name: ROM_BLOCK

Attribute subtype: string

Decorated item: constant, variable, signal

The attribute ROM_BLOCK shall indicate that the decorated item is to be implemented as a ROM. When this attribute is used to decorate a variable or signal, then all the assignments to the variable or signal must be static expressions. The value of the attribute shall indicate the name of a specific cell or module to be used to implement the decorated item. See 6.5.1 for examples of the ROM_BLOCK attribute.

7.1.5.2 RAM

Attribute name: RAM_BLOCK

Attribute subtype: string

Decorated item: variable, signal

The attribute RAM_BLOCK shall indicate that the decorated item is to be implemented as a RAM. A non-null value of the attribute shall indicate the name of a specific cell or module to be be used to implement the decorated item. See 6.5.2 for examples of the ROM_BLOCK attribute.

7.1.5.3 Logic block

Attribute name: LOGIC_BLOCK

Attribute subtype: boolean

Decorated item: constant, variable, signal

The attribute LOGIC_BLOCK shall indicate that the decorated item is to be implemented either as random logic (as opposed to a ROM implementation) or discrete sequential logic (as opposed to a RAM implementation).

7.1.6 Combinational logic attribute

Attribute name: COMBINATIONAL

Attribute subtype: boolean

Decorated item: process, conditional signal assignment, selected assignment

The COMBINATIONAL attribute shall indicate to the synthesis tool that the decorated item implies only combinational logic. Decoration by COMBINATIONAL of an object modeling sequential logic shall be an error except in the one case of a latch model in which the feedback path is defined by identity statements; such a latch shall be synthesized as combinational logic with a feedback path.

NOTE—When this attribute is TRUE, it represents the intent to prevent latches; so combinational logic with a feedback path should be synthesized as a multiplexer instead.

Example:

```
attribute COMBINATIONAL of P1: process is TRUE;
P1 : process ( ENABLE, D)
     begin
        if ENABLE = '1' then
           Q <= D;
        else
           Q <= Q ;   -- Identity assignment for Q
        end if;
     end process;
```


In the example above, there will be a feedback path from Q to itself through a multiplexer. If the process above is decorated by a COMBINATIONAL attribute with value FALSE, the synthesis tool shall treat the identity assignment as a null statement and infer sequential logic (a latch in this case) from the process. See 6.2.1.1 for an example that uses the COMBINATIONAL attribute.

7.1.7 Gated clock attribute

Attribute name: GATED_CLOCK

Attribute subtype: boolean

Decorated item: signal, process

The GATED_CLOCK attribute shall indicate to the synthesis tool that the clock and the enable signal of the inferred edge-sensitive sequential element shall be combined logically to obtain the gated clock. If used to decorate a signal, the attribute shall be propagated to all ports connected to that signal. If used to decorate a process, the attribute only shall apply to the clock signal synthesized within the scope of that process. The resulting gated clock shall be supported both as a clock and as a data driver of all logic to which it is connected.

Example 1: External gating

```
-- Separate AND gate to create gated clock which
-- is then distributed to the "clock" pin of a FF
-- and to any other pin connected:
use ieee.rtl_attributes.all; -- declaration of GATED_CLOCK
attribute GATED_CLOCK of gclk : signal is true;

gclk <= clk and enable ;
process (gclk)
begin
  if rising_edge(gclk) then
    data <= data_in ;
  end if ;
end process ;
```

Example 2: Internal gating

```
-- Implied usage of gated clock for FFs with an enable pin:
use ieee.rtl_attributes.all; -- declaration of GATED_CLOCK
attribute GATED_CLOCK of GATOR2 : label is true;

GATOR2: process(clk)
       begin
         if rising_edge(clk) then
           if enable = '1' then
             data <= data_in ;  -- clk becomes a gated clock
           end if ;
         end if ;
      end process ;
```


Example 3: Internal gating of several clocks

```
use ieee.rtl_attributes.all; -- declaration of GATED_CLOCK
attribute GATED_CLOCK of GATOR3 : label is true;
GATOR3: process(clkHi, clkLo, enable)
       signal Andclock : bit;
       begin
         AndClock := clkHi and not clkLo;
         if rising_edge(clkHi) then   -- clkHi becomes gated
           if enable = '1' then
             data(14 downto 8) <= data_in(14 downto 8);
           end if;
         end if;
         if falling_edge(clkLo) then
           if enable = '1' then       -- clkLo becomes gated
             data(7 downto 0) <= data_in(7 downto 0);
           end if;
         end if;
         if rising_edge(Andclock)     -- Andclock is not gated
           then data(15) <= '0';
           else data(15) <= '1';
         end if;
       end process;
```

Example 4: Combined internal and external gating of a clock

```
use ieee.rtl_attributes.all; -- declaration of GATED_CLOCK
attribute GATED_CLOCK of GATOR4 : label  is true;
attribute GATED_CLOCK of gclk4  : signal is true;

  gclk4 <= clk or enable;  -- gclk4 is an external gated clock
GATOR4:                    -- which drives other processes, too.
  process (gclk4)
  begin
  if rising_edge(gclk4) then
    if enable = '1' then
      data <= data_in ; -- gclk4 also an internal gated clock.
    end if ;
  end if ;
  end process;
```

7.1.8 Enumeration encoding attribute

Attribute name: ENUM_ENCODING

Attribute subtype: string

Decorated item: type, subtype

The value of this attribute shall specify the encoding of the enumeration type literals. The attribute value shall be made up of tokens separated by one or more spaces. There shall be as many tokens as there are literals in the enumeration type, with the first token corresponding to the first enumeration literal, the second token corresponding to the second enumeration literal, and so on.

Each token shall be made up of a sequence of ‘0’ and ‘1’ characters. Character ‘0’ shall represent a logic 0 value and character ‘1’ shall represent a logic 1 value. Additionally, each token may optionally contain underscore characters; these shall be used for enhancing readability and shall be ignored. All tokens shall be composed of the same number of characters (ignoring the underscore characters). Given the following enumerated type declaration and attribute declaration:

```
type <enumeration_type> is
                    (<enum_lit1>, <enum_lit2>, ... <enum_litN>);

attribute ENUM_ENCODING: STRING; -- Attribute declaration
```

The attribute specification defines the encoding for the enumeration literals as follows:

```
attribute ENUM_ENCODING of <enumeration_type>: type is
      "[<spacer>]<token1><spacer><token2><spacer> ...
                  <tokenN>[<spacer>]";        -- Attribute specification
```

Token <token1> specifies the encoding for <enum_lit1>, <token2> specifies the encoding for <enum_lit2>, and so on. <spacer> represents one or more of the following ieee.standard.character values: HT, CR, LF, or ' '.

NOTE—Use of this attribute may lead to simulation mismatches, e.g., with use of relational operators.

Example:

```
-- Example shows ENUM_ENCODING used to describe one-hot encoding:

attribute ENUM_ENCODING: string;
type COLOR is (RED, GREEN, BLUE, YELLOW, ORANGE);

attribute ENUM_ENCODING of COLOR:
                    type is "10000 01000 00100 00010 00001";

-- Enumeration literal RED is encoded with the first value 10000,
-- GREEN is encoded with the value 01000, and so on.
```

Other attributes or specifications to define encoding shall be ignored.

7.1.9 Finite state machine attribute

Attribute name: FSM_STATE

Attribute subtype: string

Decorated item: type, subtype, signal, variable

The FSM_STATE attribute shall explicitly identify the state vector for FSM extraction during synthesis. The value of the attribute shall specify the encoding scheme to be used for encoding the state.

The value of this attribute for an FSM with N states shall be one of

— "BINARY": state encoded as binary value with LOG2(N) bits
— "GRAY": state encoded as a binary value with the restriction that exactly one bit changes during state transition
— "ONE_HOT": state encoded with N bits where each encoding has a single '1'
— "ONE_COLD": state encoded with N bits where each encoding has a single '0'
— "AUTO": state encoding selection is left to the synthesis tool
— "" : state encoding selection is left to the synthesis tool
— A string of the form specified for the ENUM_ENCODING attribute, if the decorated item is of an enumeration type or subtype

Example:

```
type STATES is (S1, S2, S3, S4);
signal STATE1, STATE2, STATE3, STATE4 : STATES;

attribute FSM_STATE of STATE1: signal is "BINARY";
          --a valid encoding is: S1 = "00", S2 = "01", S3 = "10", S4 = "11"

attribute FSM_STATE of STATE2: signal is "GRAY";
          --a valid encoding is: S1 = "00", S2 = "01", S3 = "11", S4 = "10"
          --(assuming a sequential transition of S1 -> S2 -> S3 -> S4)

attribute FSM_STATE of STATE3: signal is "ONE_HOT";
          --a valid encoding is: S1 = "0001", S2 = "0010", S3 = "0100",
          --                     S4 = "1000"

attribute FSM_STATE of STATE4: signal is "ONE_COLD";
          --a valid encoding is: S1 = "1110", S2 = "1101", S3 = "1011",
          --                     S4 = "0111"
```

NOTE—In the special case in which the decorated signal is of an enumerated type, then the FSM_STATE directive can be used in a manner similar to the the ENUM_ENCODING attribute (with the difference that the latter is applied to the enumeration type). Such usage combines the identification of the state vector and a user-specified encoding in the same attribute—no additional ENUM_ENCODING attribute is then required. In addition, it allows different finite state machines with the same set of states but different transition sequences to have different state encodings.

Example:

```
type STATES is (S1, S2, S3, S4);
signal STATE : STATES;
attribute FSM_STATE of STATE: signal is "0110 0111 0000 1010";
```

If the decorated signal or variable is of an enumerated type, then the FSM_STATE attribute shall take precedence over any ENUM_ENCODING attribute specified for the enumerated type.

NOTE—Simulation mismatches may occur with the use of this attribute when a value other than binary encoding is used.

7.1.10 Finite state machine completion attribute

Attribute name: FSM_COMPLETE

Attribute subtype: boolean

Attributed object: signal, variable, type, subtype

The FSM_COMPLETE attribute shall decorate an item that represents the state register of a finite-state machine.

If the attribute value is TRUE, those states in the synthesized machine for which no transition is specified in the VHDL source shall transition to the state specified by the VHDL default state assignment. The default state assignment is the value that would be assigned to the state register if the process was executed with an invalid or unused value of the state type.

NOTES

1—FSM_COMPLETE augments the state machine hardware with transitions that allow it to recover if an invalid or unused state value occurs, as might happen because of a power glitch or single-event upset.

2—Typical ways to make a default state assignment are by the **others** clause of a case statement, the else clause of an if statement, or by an initializing value unconditionally assigned to the state register.

It shall be an error if an item is decorated with the FSM_COMPLETE attribute, the attribute value is TRUE, and there is not a unique default state assignment. It shall be an error if an item is decorated with the FSM_COMPLETE attribute, the attribute value is TRUE, and the statemachine VHDL source defines unreachable states.

NOTE—VHDL RTL and gate level simulations will match for all values of the state register in the VHDL.

Example:

```
type StateType is (S0, S1, S2, S3, S4);
signal state, next: StateType;
attribute FSM_STATE of state : signal is
             "0000 0011 0110 1100 1001" ;
attribute FSM_COMPLETE of state : signal is TRUE;

. . .

StateProc : process
begin
  wait until Clk = '1' ;
  if nReset = '1' then
     state <= S0
  else
     case state is
       when S0 =>      state <= S1;
       when S1 =>      state <= S2;
       when S2 =>      state <= S3;
       when S3 =>      state <= S4;
```

```
            when S4 =>       state <= S0;
            when others =>  state <= S0;
          end case;
       end if  ;
    end process;
```

In the example above, the VHDL specification contains five state values: S0, S1, S2, S3, and S4. FSM_STATE specifies the encoding to be a four bit array with S0 = 0000, S1 = 0011, S2 = 0110, S3 = 1100, and S4 = 1001. The implementation contains 2**4 states = 16. There are 11 states in the implementation that are not part of the VHDL specification. As FSM_COMPLETE is true, the transition for the 11 unused states is to the state specified in the others clause.

NOTE—The use of both FSM_COMPLETE TRUE and FSM_STATE ONE_HOT can incur a significant amount of logic to effect the recovery transitions. For a safe state machine, rather than using ONE_HOT, it is recommended to specify enumerated values with a hamming distance of two between them (as shown in the example).

7.1.11 Buffering attribute

Attribute name: BUFFERED

Attribute subtype: string

Decorated item: signal

The BUFFERED attribute shall be used to identify signals requiring special or high drive buffers (such as clock and reset). The value of the attribute shall identify the technology cell that shall be used to drive the signal or it shall be one the following values:

— "HIGH_DRIVE": Select a high drive buffer from the synthesis library
— "CLOCK_BUF": Select a clock buffer from the synthesis library
— "RESET_BUF": Select a reset buffer from the synthesis library

Example:

```
    attribute BUFFERED of MYCLK : signal is "CLKBUFx4";
```

In the example above, signal MYCLK wired to the input pin of the buffer cell CLKBUFx4, and all of the elements that were originally driven by MYCLK, will be driven by the output pin of the clock buffer CLKBUFx4.

The overall effect of the buffer insertion shall be noninverting. If an inverting buffer is specified by this attribute, then additional inverting logic shall be wired to the input of the buffer specified so as not to change the polarity of the signal.

NOTE—After placing the buffer, the synthesis tool is permitted to do any optimization permitted by the synthesis library.

7.2 Metacomments

Two metacomments shall be provided for conditional synthesis control. They shall be

a) -- RTL_SYNTHESIS OFF (or, abbreviated, -- RTL_SYN OFF)
b) -- RTL_SYNTHESIS ON (or, abbreviated, -- RTL_SYN ON)

A synthesis tool shall ignore any VHDL code after an "RTL_SYNTHESIS OFF" metacomment and before the first subsequent "RTL_SYNTHESIS ON" metacomment.

Metacomments differing only in the use of corresponding uppercase and lowercase letters shall be considered the same. Unabbreviated and abbreviated forms of these metacomments shall be considered the same. Whitespace shall be allowed between the line-comment token "--" and the comment.

The source code as a whole, including ignored constructs, shall conform to IEEE Std 1076-2002. The source code exclusive of constructs ignored because of the metacomments shall be compliant to the terms of this standard.

NOTES

1—Care should be taken when using these metacomments to ensure that synthesis behavior accurately reflects simulation behavior. Use of these metacomments may lead to simulation mismatches.

2—The interpretation of comments other than RTL_SYNTHESIS OFF and RTL_SYNTHESIS ON by a synthesis tool is not compliant with this standard.

8. Syntax

NOTE—Subclause titles in this clause match those of IEEE Std 1076-2002.

8.1 Design entities and configurations

8.1.1 Entity declarations

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

Supported:

- entity_declaration
- entity_declarative_part
- reserved word entity after reserved word end

Ignored:

- entity_statement_part

Example:

```
library IEEE;
use IEEE.std_Logic_1164.all;

entity E is
generic(DEPTH : Integer := 8);
  port ( CLOCK     : in    std_logic;
         RESET     : in    std_logic;
             A     : in    std_logic_vector(7 downto 0);
             B     : inout std_logic_vector(7 downto 0);
             C     : out   std_logic_vector(7 downto 0));
end E;
```

8.1.1.1 Entity header

```
entity_header ::=
[ formal_generic_clause ]
[ formal_port_clause ]

generic_clause ::= generic( generic_list );

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

Supported:

— entity_header
— generic_clause
— port_clause

a) Generics

```
generic_list ::= generic_interface_list
```

Supported:

— generic_list.

b) Ports

```
port_list ::= port_interface_list
```

Supported:

— port_list

8.1.1.2 Entity declarative part

entity_declarative_part ::=
  { entity_declarative_item }

entity_declarative_item ::
    subprogram_declaration
  | subprogram_body
  | type_declaration
  | subtype_declaration
  | constant_declaration
  | signal_declaration
  | ~~*shared*_variable_declaration~~
  | file_declaration
  | alias_declaration
  | attribute_declaration
  | attribute_specification
  | ~~disconnection_specification~~
  | use_clause
  | ~~group_template_declaration~~
  | ~~group_declaration~~

Supported:

- entity_declarative_part
- entity_declarative_item

Ignored:

- file_declaration

Not Supported:

- shared_variable_declaration
- disconnection_specification
- group_template_declaration
- group_declaration

8.1.1.3 Entity statement part

entity_statement_part ::=
{ entity_statement }

entity_statement ::=
 concurrent_assertion_statement
 | *passive*_concurrent_procedure_call
 | *passive*_process_statement

Ignored:

- entity_statement_part
- entity_statement

NOTE—The entity statement part describes passive behavior for simulation monitoring purposes. It cannot drive signals in the architecture. It, therefore, has no effect on the behavior of the architecture.

8.1.2 Architecture bodies

```
architecture_body ::=
architecture identifier of entity_name is
 architecture_declarative_part
  begin
    architecture_statement_part
  end [ architecture ] [ architecture_simple_name ] ;
```

Supported:

— architecture_body
— Multiple architectures
— Reserved word architecture after reserved word end

Not Supported:

— Global signal interactions between synthesized architectures

8.1.2.1 Architecture declarative part

```
architecture_declarative_part ::=
  { block_declarative_item }

block_declarative_item ::=
      subprogram_declaration
   |  subprogram_body
   |  type_declaration
   |  subtype_declaration
   |  constant_declaration
   |  signal_declaration
   |  shared variable_declaration
   |  file_declaration
   |  alias_declaration
   |  component_declaration
   |  attribute_declaration
   |  attribute_specification
   |  configuration_specification
   |  disconnection_specification
   |  use_clause
   |  group_template_declaration
   |  group_declaration
```

Supported:

— architecture_declarative_part
— block_declarative_item

Ignored:

— file_declaration

Not Supported:

— shared_variable_declaration
— disconnection_specification
— group_template_declaration
— group_declaration

8.1.2.2 Architecture statement part

```
architecture_statement_part ::=
  { concurrent_statement }
```

Supported:

— architecture_statement_part, as discussed in 8.9

8.1.3 Configuration declaration

```
configuration_declaration ::=
  configuration identifier of entity_name is
    configuration_declarative_part
    block_configuration
  end [configuration] [configuration_simple_name];

configuration_declarative_part ::=
  { configuration_declarative_item }
configuration_declarative_item ::=
    use_clause
  | attribute_specification
  | group_declaration
```

Supported:

— configuration_declaration
— configuration_declarative_part
— configuration_declarative_item

Not Supported:

— group_declaration

8.1.3.1 Block configuration

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

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

```
index_specification ::=
    discrete_range
  | static_expression

configuration_item ::=
    block_configuration
  | component_configuration
```

Supported:

— block_configuration
— block_specification
— index_specification
— configuration_item

8.1.3.2 Component configuration

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

Supported:

— component_configuration

8.2 Subprograms and packages

8.2.1 Subprogram declarations

```
subprogram_declaration ::=
  subprogram_specification ;

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

designator ::= identifier | operator_symbol

operator_symbol ::= string_literal
```

Supported:

— subprogram_declaration
— subprogram_specification
— designator
— operator_symbol

8.2.1.1 Formal parameters

```
formal_parameter_list ::= parameter_interface_list
```

Supported:

— formal_parameter_list

A subprogram shall not assign to an index or a slice of an unconstrained out parameter unless the associated actual parameter in each call to the subprogram is a static name. Synthesis shall use the default value as the "tie" value if a formal parameter of mode in is left open.

a) **Constant and variable parameters**

Constant and variable parameters shall be supported.

b) **Signal parameters**

Signal parameters shall be supported.

c) **File parameters**

File parameters shall not be supported.

8.2.2 Subprogram bodies

```
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
  | type_declaration
  | subtype_declaration
  | constant_declaration
  | variable_declaration
  | file_declaration
  | alias_declaration
  | attribute_declaration
  | attribute_specification
  | use_clause
  | group_template_declaration
  | group_declaration

subprogram_statement_part ::=
  { sequential_statement }

subprogram_kind ::= procedure | function
```

Supported:

- subprogram_body
- subprogram_declarative_part
- subprogram_declarative_item
- subprogram_statement_part

Ignored:

- file_declaration

Not Supported:

- group_template_declaration
- group_declaration

Subprogram recursion shall be supported when the number of recursions is bounded by a static value.

8.2.3 Subprogram overloading

8.2.3.1 Operator overloading

Operator overloading shall be supported.

a) **Signatures**

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

Signatures shall be supported.

8.2.4 Resolution functions

The resolution function RESOLVED is supported in subtype STD_LOGIC. All other resolution functions shall be ignored.

8.2.5 Package declarations

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

package_declarative_part ::=
  { package_declarative_item }

package_declarative_item ::=
    subprogram_declaration
  | type_declaration
  | subtype_declaration
  | constant_declaration
  | signal_declaration
  | ~~*shared*_variable_declaration~~
  | <u>file_declaration</u>
  | alias_declaration
  | component_declaration

```
    | attribute_declaration
    | attribute_specification
    | ~~disconnection_specification~~
    | use_clause
    | ~~group_template_declaration~~
    | ~~group_declaration~~
```

Supported:

- — package_declaration
- — package_declarative_part
- — package_declarative_item
- — keyword package after keyword end

Ignored:

- — file_declaration

Not Supported:

- — shared_variable_declaration
- — disconnection_specification
- — group_template_declaration
- — group_declaration

Signal declarations shall have an initial value expression.

Furthermore, a signal declared in a package shall have no sources. A constant declaration shall include the initial value expression; that is, deferred constants are not supported.

8.2.6 Package bodies

```
package_body ::=
  package body package_simple_name is
    package_body_declarative_part
  end [ package body ] [ package_simple_name ] ;

package_body_declarative_part ::=
  { package_body_declarative_item }

package_body_declarative_item ::=
    subprogram_declaration
  | subprogram_body
  | type_declaration
  | subtype_declaration
  | constant_declaration
  | ~~shared_variable_declaration~~
  | file_declaration
  | alias_declaration
  | use_clause
  | ~~group_template_declaration~~
  | ~~group_declaration~~
```

Supported:

— package_body
— package_body_declarative_part
— package_body_declarative_item
— keywords package body after keyword end

Ignored:

— file_declaration

Not Supported:

— shared_variable_declaration
— group_template_declaration
— group_declaration

8.3 Types

8.3.1 Scalar types

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

range_constraint ::= range range

range ::=
    range_attribute_name
  | simple_expression direction simple_expression

direction ::= to | downto
```

Supported:

— scalar_type_definition
— range_constraint
— range
— direction

Ignored:

— floating_type_definition

Not Supported:

— physical_type_definition

Null ranges shall not be supported.

8.3.1.1 Enumeration types

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

enumeration_literal ::= identifier | character_literal
```

Supported:

— enumeration_type_definition

— enumeration_literal

Elements of the following enumeration types (and their subtypes) shall be mapped to single bits as specified by IEEE Std 1076.3-1997:

a) BIT and BOOLEAN

b) STD_ULOGIC

The synthesis tool may select a default mapping for elements of other enumeration types. The user may override the default mapping by means of the ENUM_ENCODING attribute (see 7.1.8).

a) Predefined enumeration types

Supported:

— CHARACTER

Ignored:

— SEVERITY_LEVEL

Not Supported:

— FILE_OPEN_KIND

— FILE_OPEN_STATUS

8.3.1.2 Integer types

```
integer_type_definition ::= range_constraint
```

Supported:

— integer_type_definition

It is recommended that a synthesis tool should convert a signal or variable that has an integer subtype indication to a corresponding vector of bits. If the range contains no negative values, the item should have an unsigned binary representation. If the range contains one or more negative values, the item should have a twos-complement implementation. The vector should have the smallest width consistent with these representations.

The synthesis tool shall support integer types and positive, negative, and unconstrained (universal) integers whose bounds lie within the range –2 147 483 648 to +2 147 483 647 inclusive (the range that successfully maps 32 bit twos-complement numbers).

Subtypes NATURAL and POSITIVE are supported.

Integer ranges shall be synthesized as if the zero value is included.

Example: “INTEGER **range** 9 **to** 10” should be synthesized using an equivalent vector length of 4 bits, just as if it had been defined with a subtype indication of “INTEGER **range** 0 **to** 15”.

8.3.1.3 Physical types

```
~~physical_type_definition~~ ::=
  range_constraint
    units
      primary_unit_declaration
    { secondary_unit_declaration }
    end units [ physical_type_simple_name ]

~~primary_unit_declaration~~ ::= identifier ;

~~secondary_unit_declaration~~ ::= identifier = physical_literal;

physical_literal ::= [ abstract_literal ] unit_name
```

Ignored:

— physical_literal of type TIME within an ignored construct such as an **after** clause or in the initial-value expression of a declaration of an object of type TIME

Not Supported:

— physical_type_definition
— physical_literal, except of type TIME and occurring where ignored

Physical objects and literals other than of the predefined physical type TIME shall not be supported.

Declarations of objects of type TIME shall be ignored. References to objects and literals of type TIME may occur only within ignored constructs **after** clause.

8.3.1.4 Floating point types

```
floating_type_definition ::= range_constraint
```

Ignored:

— floating_type_definition

Floating point type declarations shall be ignored. Reference to objects and literals of a floating point type may occur only within ignored constructs, for example, after the **after** clause.

8.3.2 Composite types

```
composite_type_definition ::=
    array_type_definition
  | record_type_definition
```

Supported:

— composite_type_definition

8.3.2.1 Array types

```
array_type_definition ::=
    unconstrained_array_definition
  | constrained_array_definition

unconstrained_array_definition ::=
  array ( index_subtype_definition { , index_subtype_definition } )
    of element_subtype_indication

constrained_array_definition ::=
  array index_constraint of element_subtype_indication

index_subtype_definition ::= type_mark range <>

index_constraint ::= ( discrete_range { , discrete_range } )

discrete_range ::= discrete_subtype_indication | range

range ::= range_attribute_name |
            simple_expression direction simple_expression
```

Supported:

— array_type_definition
— unconstrained_array_definition
— constrained_array_definition
— index_subtype_definition
— index_constraint
— discrete_range

The bounds of a discrete range shall be specified directly or indirectly as static values belonging to an integer type. An element subtype indication shall denote either a subtype of an integer or enumeration type or a one-dimensional vector of an enumeration type whose elements denote single bits.

Null ranges shall not be supported.

If a discrete range is specified using a discrete subtype indication, the discrete subtype indication shall denote a subtype of an integer type.

A range shall comprise integer values.

a) **Index constraints and discrete ranges**

These shall be supported.

b) **Predefined array types**

Predefined array types shall be supported.

8.3.2.2 Record types

```
record_type_definition ::=
  record
      element_declaration
    { element_declaration }
  end record [ record_type_simple_name ]

element_declaration ::=
                  identifier_list : element_subtype_definition;

identifier_list ::= identifier { , identifier }

element_subtype_definition ::= subtype_indication
```

Supported:

- record_type_definition
- element_declaration
- identifier_list
- element_subtype_definition

8.3.3 Access types

```
access_type_definition ::= access subtype_indication
```

Ignored:

- access_type_definition

The use of objects of access type shall not be supported.

8.3.3.1 Incomplete type declarations

```
incomplete_type_declaration ::= type identifier ;
```

Ignored:

- incomplete_type_declaration

8.3.3.2 Allocation and deallocation of objects

Allocation and deallocation shall not be supported.

8.3.4 File types

```
file_type_definition ::= file of type_mark
```

Ignored:

- file_type_definition

Use of file objects (objects declared as belonging to a file type) shall not be supported.

8.3.4.1 File operations

Not Supported:

— File operations

8.4 Declarations

```
declaration ::=
    type_declaration
  | subtype_declaration
  | object_declaration
  | interface_declaration
  | alias_declaration
  | architecture_body
  | attribute_declaration
  | component_declaration
  | group_template_declaration
  | group_declaration
  | entity_declaration
  | configuration_declaration
  | subprogram_declaration
  | package_declaration
  | primary_unit
```

Supported:

— declaration

Not Supported:

— group_template_declaration
— group_declaration

8.4.1 Type declarations

```
type_declaration ::=
    full_type_declaration
  | incomplete_type_declaration

full_type_declaration ::=
  type identifier is type_definition ;

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

Supported:

— type_declaration
— full_type_declaration
— type_definition

Ignored:

— incomplete_type_declaration
— access_type_definition
— file_type_definition

Full type declarations containing access type definitions or file type definitions shall be ignored.

Not Supported:

— protected_type_definition

8.4.2 Subtype declarations

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

subtype_indication ::=
  [ resolution_function_name ] type_mark [ constraint ]

type_mark ::=
    type_name
  | subtype_name

constraint ::=
    range_constraint
  | index_constraint
```

Supported:

— subtype_declaration
— subtype_indication
— type_mark
— constraint

Ignored:

— User-defined resolution functions

8.4.3 Objects

8.4.3.1 Object declarations

```
object_declaration ::=
    constant_declaration
  | signal_declaration
  | variable_declaration
  | file_declaration
```

Supported:

— object_declaration

Ignored:

— file_declaration

a) Constant declarations

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

Supported:

— constant_declaration

Deferred constant declaration shall not be supported. That is, the expression shall be present in the constant declaration.

b) Signal declarations

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

signal_kind ::= register | bus
```

Supported:

— signal_declaration

— signal_kind

Ignored:

— expression

The initial value expression shall be ignored unless the declaration is in a package, where the declaration shall have an initial value expression.

The subtype indication shall be a globally static type. An assignment to a signal declared in a package shall not be supported.

c) Variable declarations

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

Supported:

— variable_declaration

Ignored:

— expression

Not Supported:

— Reserved word shared

The reserved word shared shall not be supported. The initial value expression shall be ignored. The subtype indication shall be a globally static type.

The use of access objects shall not be supported.

d) File declarations

```
file_declaration ::=
```

```
  file identifier_list :
                  subtype_indication [ file_open_information ] ;

file_open_information ::=
  [ open file_open_kind_expression ] is file_logical_name

file_logical_name ::= string_expression
```

Ignored:

— file_declaration

The use of file objects shall not be supported.

8.4.3.2 Interface declarations

```
interface_declaration ::=
    interface_constant_declaration
  | interface_signal_declaration
  | interface_variable_declaration
  | ~~interface_file_declaration~~

interface_constant_declaration ::=
  [constant] identifier_list :
                   [in] subtype_indication [:= static_expression]

interface_signal_declaration ::=
  [signal] identifier_list : [mode] subtype_indication [bus]
    [:= static_expression]

interface_variable_declaration ::=
  [variable] identifier_list : [mode] subtype_indication
    [:= static_expression]

~~interface_file_declaration~~ ::=
  file identifier_list : subtype_indication

mode ::= in | out | inout | buffer | ~~linkage~~
```

Supported:

— interface_declaration
— interface_constant_declaration
— interface_signal_declaration
— interface_variable_declaration

Ignored:

— static_expression (interface signal declarations and interface variable declarations).

Not Supported:

— interface_file_declaration
— mode linkage

The static expression shall be ignored in port interface lists and formal parameter lists.

Static expressions in interface constant declarations shall be supported.

a) **Interface lists**

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

interface_element ::= interface_declaration
```

Supported:

— interface_list
— interface_element

b) **Association lists**

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

association_element ::=
  [formal_part =>] actual_part

formal_part ::=
    formal_designator
  | function_name( formal_designator )
  | type_mark( formal_designator )

formal_designator ::=
    generic_name
  | port_name
  | parameter_name

actual_part ::=
    actual_designator
  | function_name( actual_designator )
  | type_mark( actual_designator )

actual_designator ::=
    expression
  | signal_name
  | variable_name
  | file_name
  | open
```

Supported:

— association_list
— association_element
— formal_part
— formal_designator
— actual_part
— actual_designator

Not Supported:

— file_name

8.4.3.3 Alias declarations

```
alias_declaration ::=
  alias alias_designator [: subtype_indication]
                                          is name [signature];

alias_designator ::= identifier
                           | character_literal | operator_symbol
```

Supported:

— alias_declaration

— alias_designator

8.4.4 Attribute declarations

```
attribute_declaration ::=
  attribute identifier : type_mark ;
```

Supported:

— attribute_declaration

Ignored:

— User-defined attribute declarations other than those of the synthesis-specific attributes in this standard

8.4.5 Component declarations

```
component_declaration ::=
  component identifier [is]
    [local_generic_clause]
    [local_port_clause]
  end component [component_simple_name] ;
```

Supported:

— component_declaration

8.4.6 Group template declarations

```
group_template_declaration ::=
  group identifier is ( entity_class_entry_list ) ;

entity_class_entry_list ::=
  entity_class_entry {, entity_class_entry }

entity_class_entry ::= entity_class [<>]
```


Not Supported:

— group_template_declaration
— entity_class_entry_list
— entity_class_entry

8.4.7 Group declarations

```
~~group_declaration~~ ::=
  group identifier : group_template_name( group_consituent_list );

~~group_constituent_list~~ ::= group_constituent {, group_constituent }

~~group_constituent~~ ::= name | character_literal
```

Not Supported:

— group_declaration
— group_constituent_list
— group_constituent

8.5 Specifications

8.5.1 Attribute specification

```
attribute_specification ::=
  attribute attribute_designator
                           of entity_specification is expression;

entity_specification ::=
  entity_name_list : entity_class

entity_class ::=
  entity    | architecture | configuration
| procedure | function     | package
| type      | subtype      | constant
| signal    | variable     | component
| label     | literal      | units
| ~~group~~     | ~~file~~

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

entity_designator ::= entity_tag [signature]

entity_tag ::= simple_name | character_literal | operator_symbol
```

Supported:

- attribute_specification
- entity_specification
- entity_class
- entity_name_list
- entity_designator
- entity_tag

Ignored:

- User-defined attribute declarations and their specifications, except those of the synthesis-specific attributes of Clause 7

Not Supported:

- entity class group and file
- reading of names of user-defined attributes

8.5.2 Configuration specification

```
configuration_specification ::=
  for component_specification binding_indication;

component_specification ::=
  instantiation_list : component_name

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

Supported:

- configuration_specification
- component_specification
- instantiation_list

8.5.2.1 Binding indication

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

Supported:

- binding_indication

a) **Entity aspect**

```
entity_aspect ::=
    entity entity_name [(architecture_identifier)]
      | configuration configuration_name
      | open
```


Supported:

— entity_aspect

b) Generic map and port map aspects

```
generic_map_aspect ::=
  generic map ( generic_association_list )

port_map_aspect ::=
  port map ( port_association_list )
```

8.5.2.2 Default binding indication

Default binding shall be supported.

8.5.3 Disconnection specification

Disconnection specifications shall not be supported.

8.6 Names

8.6.1 Names

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

prefix ::=
    name
  | function_call
```

Supported:

— name

— prefix

8.6.2 Simple names

```
simple_name ::= identifier:
```

Supported:

— simple_name

8.6.3 Selected names

```
selected_name ::= prefix.suffix

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

Supported:
— selected_name
— suffix

8.6.4 Indexed names

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

Supported:
— indexed_name

Using an indexed name of an unconstrained out parameter in a procedure shall not be supported.

8.6.5 Slice names

```
slice_name ::= prefix ( discrete_range )
```

Supported:
— slice_name

Using a slice name of an unconstrained out parameter in a procedure shall not be supported.

Null slices shall not be supported.

For a discrete range that appears as part of a slice name, the bounds of the discrete range shall be specified directly or indirectly as static values belonging to an integer type.

8.6.6 Attribute names

```
attribute_name ::=
  prefix [signature]'attribute_designator [ ( expression ) ]

attribute_designator ::= attribute_simple_name
```

Supported attribute designators:
— 'BASE
— 'LEFT
— 'RIGHT
— 'HIGH
— 'LOW
— 'RANGE

— 'REVERSE_RANGE
— 'LENGTH
— 'EVENT
— 'STABLE

Supported:

— attribute_name
— attribute_designator

Attributes 'EVENT and 'STABLE shall only be used as specified in 6.1.

8.7 Expressions

8.7.1 Expressions

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

relation ::=
  shift_expression [ relational_operator shift_expression ]

shift_expression ::=
  simple_expression [ shift_operator simple_expression ]

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

term ::=
  factor { multiplying_operator factor }

factor ::=
    primary [ ** primary ]
  | abs primary
  | not primary

primary ::=
    name
  | literal
  | aggregate
  | function_call
  | qualified_expression
  | type_conversion
  | ~~allocator~~
  | ( expression )
```

Supported:

- — expression
- — relation
- — shift_expression
- — simple_expression
- — term
- — factor
- — primary

Not Supported:

- — allocator in a primary

8.7.2 Operators

```
      logical_operator ::= and | or  | nand | nor | xor | xnor
   relational_operator ::=  =  | /=  |  <   | <=  |  >  | >=
        shift_operator ::= sll | srl |  sla | sra | rol | ror
       adding_operator ::=  +  |  -  | &
                  sign ::=  +  |  -
  multiplying_operator ::=  *  |  /  | mod  | rem
miscellaneous_operator ::= **  | abs | not
```

Supported:

- — logical_operator
- — relational_operator
- — adding_operator
- — sign
- — multiplying_operator
- — miscellaneous_operator

8.7.2.1 Logical operators

No restriction.

8.7.2.2 Relational operators

No restriction.

NOTE—Using relational operators for an enumerated type that has an explicit encoding specified via the ENUM_ENCODING attribute may lead to simulation mismatches (see 7.1.8).

8.7.2.3 Shift operators

No restriction.

8.7.2.4 Adding operators

No restriction.

8.7.2.5 Sign operators

No restriction.

8.7.2.6 Multiplying operators

Supported:

— * (multiply) operator

— / (division), mod, and rem operators

— all multiplying operators defined in IEEE Std 1076.3-1997

8.7.2.7 Miscellaneous operators

Supported:

— ** (exponentiation) operator

— **abs** operator

The ** (exponentiation) operator shall be supported only when both operands are static or when the left operand has the static value 2.

8.7.3 Operands

8.7.3.1 Literals

```
literal ::=
    numeric_literal
      | enumeration_literal
      | string_literal
      | bit_string_literal
      | null
```

(`null` is struck through.)

```
numeric_literal ::=
    abstract_literal
  | physical_literal.
```

(`physical_literal` is struck through.)

Supported:

— literal

— numeric_literal

Not Supported:

— physical literal, except of type TIME occurring where ignored

— null

Physical literals of type TIME and floating point literals may occur only within ignored constructs, for example, **after** clauses.

8.7.3.2 Aggregates

```
aggregate ::=
  ( element_association {, element_association} )

element_association ::=
  [ choices => ] expression

choices ::= choice { | choice }

choice ::=
    simple_expression
  | discrete_range
  | element_simple_name
  | others
```

Supported:

- aggregate
- element_association
- choices
- choice
- Use of a type as a choice

Example:

```
subtype  Src_Typ  is Integer range 7 downto 4;
subtype  Dest_Typ is Integer range 3 downto 0;
-- Constant definition with aggregates
constant Data_c : Std_Logic_Vector(7 downto 0)
                           := (Src_Typ  => '1', Dest_Typ => '0');
```

a) Record aggregates

No restriction.

b) Array aggregates

No restriction.

8.7.3.3 Function calls

```
function_call ::=
  function_name [ ( actual_parameter_part ) ]

actual_parameter_part ::= parameter_association_list
```

Supported:

- function_call
- actual_parameter_part

Restrictions exist for the actual parameter part. These restrictions are described in 8.4.3.2.

8.7.3.4 Qualified expressions

```
qualified_expression ::=
    type_mark'( expression )
  | type_mark'aggregate
```

Supported:

— qualified_expression

8.7.3.5 Type conversions

```
type_conversion ::= type_mark( expression )
```

Supported:

— type_conversion

8.7.3.6 Allocators

```
allocator ::=
    new subtype_indication
  | new qualified_expression
```

Not Supported:

— allocator

8.7.4 Static expressions

8.7.4.1 Locally static primaries

Locally static primaries shall be supported.

8.7.4.2 Globally static primaries

Globally static primaries shall be supported.

8.7.5 Universal expressions

Floating-point expressions shall not be supported. Infinite-precision expressions shall not be supported. Precision shall be limited to 32 bits.

8.8 Sequential statements

```
sequence_of_statements ::=
  { sequential_statement }

sequential_statement ::=
    wait_statement
  | assertion_statement
  | report_statement
  | signal_assignment_statement
  | variable_assignment_statement
  | procedure_call_statement
```

```
        | if_statement
        | case_statement
        | loop_statement
        | next_statement
        | exit_statement
        | return_statement
        | null_statement
```

Supported:

- sequence_of_statements
- sequential_statement

8.8.1 Wait statement

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

sensitivity_clause ::= on sensitivity_list

sensitivity_list ::= signal_name {, signal_name}

condition_clause ::= until condition

condition ::= boolean_expression

timeout_clause ::= for time_expression
```

Supported:

- wait_statement
- condition_clause
- condition
- sensitivity_clause
- sensitivity_list

Ignored:

- timeout_clause

Wait statements shall be supported only to define sequential behavior. See 6.1.3.2 and 6.1.3.4 for the modeling rules applied to wait statements.

NOTE—The use of a timeout clause may lead to simulation mismatches.

8.8.2 Assertion statement

```
assertion_statement ::= [ label: ] assertion ;

assertion ::=
  assert condition
    [ report expression ]
    [ severity expression ]
```

Ignored:

— assertion_statement

— assertion

8.8.3 Report statement

```
report_statement ::=
  [label:] report expression
    [severity expression] ;
```

Ignored:

— report_statement

8.8.4 Signal assignment statement

```
signal_assignment_statement ::=
  [ label: ] target <= [ delay_mechanism ] waveform ;

delay_mechanism ::=
    transport
  | [ reject time_expression ] inertial

target ::=
    name
  | aggregate

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

Supported:

— signal_assignment_statement

— target

— waveform

— reserved word unaffected

Ignored:

— delay_mechanism

Not Supported:

— time_expression

— multiple waveform_elements

An assignment to a signal declared in a package shall not be supported.

8.8.4.1 Updating a projected output waveform

```
waveform_element ::=
    value_expression [after time_expression]
      | null [after time_expression]
```

Supported:

— waveform_element

Ignored:

— time expression after reserved word **after**

Not Supported:

— null waveform elements

8.8.5 Variable assignment statement

```
variable_assignment_statement ::=
  [ label: ] target := expression ;
```

Supported:

— variable_assignment_statement

8.8.6 Procedure call statement

```
procedure_call_statement ::= [ label: ] procedure_call ;

procedure_call ::= procedure_name [ ( actual_parameter_part ) ]
```

Supported:

— procedure_call_statement

— procedure_call

Restrictions for the actual parameter part are described in 8.4.3.2, item b).

8.8.7 If statement

```
if_statement ::=
  [ if_label: ]
    if condition then
      sequence_of_statements
  { elsif condition then
      sequence_of_statements }
  [ else
      sequence_of_statements ]
    end if [ if_label ] ;
```

Supported:

— if_statement

NOTE—If a signal or variable is assigned under some values of the conditional expressions in the **if** statement, but not for all values, storage elements may result; see 6.1 and 6.2.

8.8.8 Case statement

```
case_statement ::=
  [ case_label: ]
    case expression is
      case_statement_alternative
    { case_statement_alternative }
    end case [ case_label ] ;

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

Supported:

— case_statement
— case_statement_alternative

NOTE—If a signal or variable is assigned under some values of the conditional expressions in the **case** statement, but not for all values, storage elements may result; see 6.1 and 6.2.

If a metalogical or high-impedance value occurs as a choice, or as an element of a choice, in a case statement that is interpreted by a synthesis tool, the synthesis tool shall ignore that choice and synthesize the case statement as though that choice did not exist. That is, the interpretation that is generated shall not be required to contain any construct corresponding to the presence or absence of the sequence of statements associated with the choice.

NOTES

1—If the type of the case expression includes metalogical or high-impedance values, and if not all of the metalogical or high-impedance values are included among the case choices, then the case statement must include an **others** choice to cover the missing metalogical or high-impedance choice values (IEEE Std 1076-2002).

2—A case choice including a metalogical or high-impedance value such as “1X1” indicates a branch that never can be taken by the synthesized circuit (IEEE Std 1076.3-1997).

8.8.9 Loop statement

```
loop_statement ::=
  [ loop_label: ]
    [ iteration_scheme ] loop
      sequence_of_statements
    end loop [ loop_label ] ;

iteration_scheme ::=
    while condition
  | for loop_parameter_specification

parameter_specification ::=
  identifier in discrete_range

discrete_range ::= discrete_subtype_indication | range
```

Supported:

— loop_statement

— iteration_scheme

— parameter_specification

— discrete_range

Not Supported:

— while

For a discrete range that appears as part of a parameter specification, the bounds of the discrete range shall be specified directly or indirectly as static values belonging to an integer type.

8.8.10 Next statement

```
next_statement ::=
  [ label: ] next [ loop_label ] [ when condition ] ;
```

Supported:

— next_statement

8.8.11 Exit statement

```
exit_statement ::=
  [ label: ] exit [ loop_label ] [ when condition ] ;
```

Supported:

— exit_statement

8.8.12 Return statement

```
return_statement ::=
  [ label: ] return [ expression ] ;
```

Supported:

— return_statement

8.8.13 Null statement

```
null_statement ::=
  [ label: ] null ;
```

Supported:

— null_statement

8.9 Concurrent statements

```
concurrent_statement ::=
    block_statement
  | process_statement
  | concurrent_procedure_call_statement
  | concurrent_assertion_statement
  | concurrent_signal_assignment_statement
  | component_instantiation_statement
  | generate_statement
```

Supported:

— concurrent_statement

8.9.1 Block statement

```
block_statement ::=
  block_label:
    block [ ( guard_expression ) ] [ is ]
      block_header
      block_declarative_part
    begin
      block_statement_part
    end block [ block_label ] ;

block_header ::=
  [ generic_clause
  [ generic_map_aspect ;] ]
  [ port_clause
  [ port_map_aspect ;] ]

block_declarative_part ::=
  { block_declarative_item }

block_statement_part ::=
  { concurrent_statement }
```

Supported:

— block_statement

— block_declarative_part

— block_statement_part

Not Supported:

— block_header

8.9.2 Process statement

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

process_declarative_part ::=
  { process_declarative_item }

process_declarative_item ::=
    subprogram_declaration
  | subprogram_body
  | type_declaration
  | subtype_declaration
  | constant_declaration
  | variable_declaration
  | file_declaration
  | alias_declaration
  | attribute_declaration
  | attribute_specification
  | use_clause
  | ~~group_template_declaration~~
  | ~~group_declaration~~

process_statement_part ::=
  { sequential_statement }
```

Supported:

- process_statement
- sensitivity_list
- process_declarative_part
- process_declarative_item
- process_statement_part

Ignored:

- file_declaration
- user-defined attribute declarations and their specifications, except those of the synthesis-specific attributes of Clause 7.

Not Supported:

- reserved word postponed
- group_template_declaration
- group_declaration

The sensitivity list shall include those signals or elements of signals that are read by the process except for signals read only under control of a clock edge, as described in Clause 6.

Attribute declarations and specifications as described in 7.1 shall be the only ones supported.

Use of file objects or access objects (variables of access type) in a process shall not be supported.

8.9.3 Concurrent procedure call statement

```
concurrent_procedure_call_statement ::=
  [ label: ] [ postponed ] procedure_call ;
```

Supported:

— concurrent_procedure_call_statement

Not Supported:

— reserved word postponed

8.9.4 Concurrent assertion statement

```
concurrent_assertion_statement ::=
  [ label: ] [ postponed ] assertion ;
```

Ignored:

— concurrent_assertion_statement

Not Supported:

— Reserved word postponed

8.9.5 Concurrent signal assignment statement

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

options ::= [ guarded ] [delay_mechanism]
```

Supported:

— concurrent_signal_assignment_statement

— guarded in options

Not Supported:

— reserved word **postponed**

Any **after** clause shall be ignored.

Multiple waveform elements shall not be supported.

Example:

```
architecture A of E is
begin

      B(7) <= A(6);
      B(3 downto 0) <= A(7 downto 4);

      C <= not A;
end A;
```

8.9.5.1 Conditional signal assignment

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

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

NOTE—Options, **guarded**, and delay_mechanism are expanded in 8.9.5.

Supported:

- conditional_signal_assignment
- conditional_waveforms

Ignored:

- delay_mechanism

A conditional waveform that contains a reference to one or more elements of the target signal shall not be supported.

Example:

```
architecture A of E is
  begin
    C <=     B          when A(0) = '1' else
         not B          when A(1) = '1' else
       "00000000"       when A(2) = '1' and RESET = '1' else
       (others => ('1'));
  end A;
```

8.9.5.2 Selected signal assignment

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

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

NOTE—Options, **guarded**, and delay_mechanism are expanded in 8.9.5.

Supported:

— selected_signal_assignment
— select_waveforms

Ignored:

— delay_mechanism

A conditional waveform that contains a reference to one or more elements of the target signal shall not be supported.

Example:

```
architecture A of E is
  begin
  with A select
    C <=                 B    when "00000000",
                     not B    when "10101010",
            (others => ('1')) when "11110001",
                     not A    when others;
  end A;
```

8.9.6 Component instantiation statement

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

instantiated_unit ::=
    [component] component_name
  | entity entity_name [( architecture_name )]
  | configuration configuration_name
```

Supported:

— component_instantiation_statement
— instantiated_unit

Restrictions exist for the generic map aspect and the port map aspect; these are described in 8.4.3.2.

8.9.6.1 Instantiation of a component

No restriction.

8.9.6.2 Instantiation of a design entity

No restriction.

8.9.7 Generate statement

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

generation_scheme ::=
    for generate_parameter_specification
  | if condition

label ::= identifier
```

Supported:

- generate_statement
- generate_scheme
- label

The generate parameter specification shall be statically computable and of the form "identifier in range" only.

8.10 Scope and visibility

8.10.1 Declarative region

Declarative regions shall be supported.

8.10.2 Scope of declarations

The scope of declarations shall be supported.

8.10.3 Visibility

Visibility rules shall be supported.

8.10.4 Use clause

```
use_clause ::=
  use selected_name {, selected_name} ;
```

The use clause shall contain only the selected name of a package.

Supported:

- use_clause

8.10.5 The context of overloaded resolution

The context of overloaded resolution shall be supported.

8.11 Design units and their analysis

8.11.1 Design units

```
design_file ::= design_unit { design_unit }

design_unit ::= context_clause library_unit

library_unit ::=
    primary_unit
  | secondary_unit

primary_unit ::=
    entity_declaration
  | configuration_declaration
  | package_declaration

secondary_unit ::=
    architecture_body
  | package_body
```

Supported:

- design_file
- design_unit
- library_unit
- primary_unit
- secondary_unit

8.11.2 Design libraries

```
library_clause ::= library logical_name_list ;

logical_name_list ::= logical_name {, logical_name}

logical_name ::= identifier
```

Supported:

- library_clause
- logical_name_list
- logical_name

8.11.3 Context clauses

```
context_clause ::= { context_item }

context_item ::=
    library_clause
    | use_clause
```

Supported:

— context_clause
— context_item

8.11.4 Order of analysis

The order of analysis shall be supported.

8.12 Elaboration

Elaboration shall not be constrained.

8.13 Lexical elements

Extended identifiers shall not be supported.

8.14 Predefined language environment

8.14.1 Predefined attributes

8.14.1.1 Attributes whose prefix type is a type t

```
t'BASE

t'LEFT

t'RIGHT

t'HIGH

t'LOW
```

~~t'ASCENDING~~

~~t'IMAGE~~

```
t'VALUE(x)

t'POS(x)

t'VAL(x)

t'SUCC(x)

t'PRED(x)

t'LEFTOF(x)

t'RIGHTOF(x)
```


8.14.1.2 Attributes whose prefix is an array object a, or attributes of a constrained array subtype a

a'LEFT[~~(n)~~]

a'RIGHT[~~(n)~~]

a'HIGH[~~(n)~~]

a'LOW[~~(n)~~]

a'RANGE[~~(n)~~]

a'REVERSE_RANGE[~~(n)~~]

a'LENGTH[~~(n)~~]

~~a'ASCENDING[(n)]~~

8.14.1.3 Attributes whose prefix is a signal s

~~s'DELAYED[(t)]~~

s'STABLE[~~(t)~~]

~~s'QUIET~~

~~s'TRANSACTION~~

s'EVENT

~~s'ACTIVE~~

~~s'LAST_EVENT~~

~~s'LAST_ACTIVE~~

~~s'LAST_VALUE~~

~~s'DRIVING~~

~~s'DRIVING_VALUE~~

Attributes STABLE and EVENT may only be used as described in Clause 6.

8.14.1.4 Attributes whose prefix is a named object e

~~e'SIMPLE_NAME~~

~~e'INSTANCE_NAME~~

~~e'PATH_NAME~~

8.14.2 Package STANDARD

Functions in the package STANDARD shall be either supported or not supported as defined below:

Supported:

- functions with one or more arguments of type CHARACTER
- functions with one or more arguments of type STRING
- all functions whose arguments are only of type BOOLEAN
- all functions whose arguments are only of type BIT
- the following functions with arguments of type "universal integer" or of type INTEGER:

  relational operator functions

  "+", "-", "**abs**", "*"

  "/", "**mod**", and "**rem**", provided both operands are static or the second argument is a static power of two

  "**" provided the first argument is a static value of two
- all functions with an argument of type BIT_VECTOR

Ignored:

-- the attribute 'FOREIGN

Not Supported:

- functions with arguments of type SEVERITY_LEVEL
- the following functions with arguments of type "universal integer" or INTEGER:

  "/", "**mod**", and "**rem**" when neither operand is static or the second argument is not a static power of two

  "**" when the first argument is not a static value of two
- functions with arguments of type "universal real" or of type REAL
- functions with one or more arguments of type TIME
- the function NOW
- functions with one or more arguments of type FILE_OPEN_KIND
- functions with one or more arguments of type FILE_OPEN_STATUS

8.14.3 Package TEXTIO

The subprograms defined in package TEXTIO shall not be supported.

Annex A

(informative)

Syntax summary

This annex summarizes the VHDL syntax that is supported.

abstract_literal ::= decimal_literal | based_literal

access_type_definition ::= **access** subtype_indication

actual_designator ::=
    expression
  | *signal*_name
  | *variable*_name
  | ~~*file*_name~~
  | **open**

actual_parameter_part ::= *parameter*_association_list

actual_part ::=
    actual_designator
  | *function*_name(actual_designator)
  | type_mark(actual_designator)

adding_operator ::= + | - | &

aggregate ::=
  (element_association {, element_association})

alias_declaration ::=
  alias alias_designator [: subtype_indication]
      is name [signature];

alias_designator ::= identifier | character_literal
      | operator_symbol

~~allocator~~ ::=
    new subtype_indication
  | **new** qualified_expression

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

architecture_declarative_part ::=
  { block_declarative_item }

architecture_statement_part ::=
{ concurrent_statement }

array_type_definition ::=
unconstrained_array_definition
| constrained_array_definition

assertion ::=
assert condition
[**report** expression]
[**severity** expression]

assertion_statement ::= [label:] assertion;

association_element ::=
[formal_part =>] actual_part

association_list ::=
association_element {, association_element}

attribute_declaration ::=
attribute identifier : type_mark ;

attribute_designator ::= *attribute*_simple_name

attribute_name ::=
prefix [signature]'attribute_designator [(expression)]

attribute_specification ::=
attribute attribute_designator **of** entity_specification
is expression;

base ::= integer

base_specifier ::= B | O | X

based_integer ::=
extended_digit { [underline] extended_digit }

based_literal ::=
base # based_integer [~~. based_integer~~] # [exponent]

basic_character ::=
basic_graphic_character | format_effector

basic_graphic_character ::=
upper_case_letter | digit | special_character| space_character

basic_identifier ::=
letter { [underline] letter_or_digit }

binding_indication ::=
  [**use** entity_aspect]
  [generic_map_aspect]
  [port_map_aspect]

bit_string_literal ::= base_specifier " [bit_value] "

bit_value ::= extended_digit { [underline] extended_digit }

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

block_declarative_item ::=
    subprogram_declaration
  | subprogram_body
  | type_declaration
  | subtype_declaration
  | constant_declaration
  | signal_declaration
  | ~~*shared*_variable_declaration~~
  | file_declaration
  | alias_declaration
  | component_declaration
  | attribute_declaration
  | attribute_specification
  | configuration_specification
  | ~~disconnection_specification~~
  | use_clause
  | ~~group_template_declaration~~
  | ~~group_declaration~~

block_declarative_part ::=
  { block_declarative_item }

~~block_header~~ ::=
  [generic_clause
  [generic_map_aspect ;]]
  [port_clause
  [port_map_aspect ;]]

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

```
block_statement ::=
  block_label:
    block [ ( guard_expression ) ] [ is ]
      block_header
      block_declarative_part
    begin
      block_statement_part
    end block [ block_label ] ;

block_statement_part ::=
  { concurrent_statement }

case_statement ::=
  [ case_label: ]
    case expression is
      case_statement_alternative
    { case_statement_alternative }
    end case [ case_label ] ;

case_statement_alternative ::=
  when choices =>
    sequence_of_statements

character_literal ::= ‘ graphic_character ‘

choice ::=
    simple_expression
  | discrete_range
  | element_simple_name
  | others

choices ::= choice { | choice }

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

component_declaration ::=
  component identifier [is]
    [local_generic_clause]
    [local_port_clause]
  end component [component_simple_name];

component_instantiation_statement ::=
  instantiation_label:
    instantiated_unit
      [ generic_map_aspect ]
      [ port_map_aspect ] ;

component_specification ::=
  instantiation_list : component_name
```

composite_type_definition ::=
    array_type_definition
  | record_type_definition

concurrent_assertion_statement ::=
  [label:] [~~**postponed**~~] assertion ;

concurrent_procedure_call_statement ::=
  [label:] [~~**postponed**~~] procedure_call ;

concurrent_signal_assignment_statement ::=
    [label:] [~~**postponed**~~] conditional_signal_assignment
  | [label:] [~~**postponed**~~] selected_signal_assignment

concurrent_statement ::=
    block_statement
  | process_statement
  | concurrent_procedure_call_statement
  | concurrent_assertion_statement
  | concurrent_signal_assignment_statement
  | component_instantiation_statement
  | generate_statement

condition ::= *boolean*_expression

condition_clause ::= **until** condition

conditional_signal_assignment ::=
  target <= options conditional_waveforms ;

conditional_waveforms ::=
  { waveform **when** condition **else** }
    waveform [**when** condition]

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

configuration_declarative_item ::=
    use_clause
  | attribute_specification
  | ~~group_declaration~~

configuration_declarative_part ::=
  { configuration_declarative_item }

configuration_item ::=
    block_configuration
  | component_configuration

configuration_specification ::=
  for component_specification binding_indication;

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

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

constraint ::=
    range_constraint
  | index_constraint

context_clause ::= { context_item }

context_item ::=
    library_clause
  | use_clause

decimal_literal ::= integer [. integer] [exponent]

declaration ::=
    type_declaration
  | subtype_declaration
  | object_declaration
  | interface_declaration
  | alias_declaration
  | attribute_declaration
  | component_declaration
  | ~~group_template_declaration~~
  | ~~group_declaration~~
  | entity_declaration
  | configuration_declaration
  | subprogram_declaration
  | package_declaration

<u>delay_mechanism</u> ::=
    transport
  | [**reject** *time*_expression] **inertial**

design_file ::= design_unit { design_unit }

design_unit ::= context_clause library_unit

designator ::= identifier | operator_symbol

direction ::= **to** | **downto**

~~disconnection_specification~~ ::=
  disconnect guarded_signal_specification **after** *time*_expression ;

discrete_range ::= *discrete*_subtype_indication | range

element_association ::=
  [choices =>] expression

element_declaration ::= identifier_list
  : element_subtype_definition ;

element_subtype_definition ::= subtype_indication

entity_aspect ::=
    entity *entity*_name [(*architecture*_identifier)]
  | **configuration** *configuration*_name
  | **open**

entity_class ::=
  entity | **architecture** | **configuration**
| **procedure** | **function** | **package**
| **type** | **subtype** | **constant**
| **signal** | **variable** | **component**
| **label** | **literal** | **units**
| ~~**group**~~ | ~~**file**~~

~~entity_class_entry~~ ::= entity_class [<>]

~~entity_class_entry_list~~ ::=
  entity_class_entry {, entity_class_entry }

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

entity_declarative_item ::
    subprogram_declaration
  | subprogram_body
  | type_declaration
  | subtype_declaration
  | constant_declaration
  | signal_declaration
  | ~~*shared*_variable_declaration~~
  | file_declaration
  | alias_declaration
  | attribute_declaration
  | attribute_specification
  | ~~disconnection_specification~~
  | use_clause
  | ~~group_template_declaration~~
  | ~~group_declaration~~

entity_declarative_part ::=
  { entity_declarative_item }

```
entity_designator ::= entity_tag [signature]

entity_header ::=
  [ formal_generic_clause ]
  [ formal_port_clause ]

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

entity_specification ::=
  entity_name_list : entity_class

entity_statement ::=
    concurrent_assertion_statement
  | passive_concurrent_procedure_call
  | passive_process_statement

entity_statement_part ::=
  { entity_statement }

entity_tag ::= simple_name | character_literal | operator_symbol

enumeration_literal ::= identifier | character_literal

enumeration_type_definition ::=
  ( enumeration_literal { , enumeration_literal } )

exit_statement ::=
  [ label: ] exit [ loop_label ] [ when condition ] ;

exponent ::= E [ + ] integer | E - integer

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

extended_digit ::= digit | letter

extended_identifier ::=
      \ graphic_character { graphic_character } \

factor ::=
    primary [ ** primary ]
  | abs primary
  | not primary
```

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

file_logical_name ::= *string*_expression

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

file_type_definition ::= **file of** type_mark

floating_type_definition ::= range_constraint

formal_designator ::=
    *generic*_name
  | *port*_name
  | *parameter*_name

formal_parameter_list ::= *parameter*_interface_list

formal_part ::=
    formal_designator
  | *function*_name(formal_designator)
  | type_mark(formal_designator)

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

function_call ::=
  *function*_name [(actual_parameter_part)]

generate_statement ::=
  *generate*_label:
    generation_scheme **generate**
      [{ block_declarative_item }
    begin]
      { concurrent_statement }
    end generate [*generate*_label] ;

generation_scheme ::=
    for *generate*_parameter_specification
  | **if** condition

generic_clause ::=
  generic(generic_list);

generic_list ::= *generic*_interface_list

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

graphic_character ::=
basic_graphic_character | lower_case_letter
| other_special_character

~~group_constituent~~ ::= name | character_literal

~~group_constituent_list~~ ::= group_constituent {, group_constituent }

~~group_declaration~~ ::=
group identifier : *group_template*_name(group_consituent_list);

~~group_template_declaration~~ ::=
group identifier **is** (entity_class_entry_list) ;

guarded_signal_specification ::=
*guarded*_signal_list : type_mark

identifier ::=
basic_identifier | extended_identifier

identifier_list ::= identifier { , identifier }

if_statement ::=
[*if*_label:]
if condition **then**
sequence_of_statements
{ **elsif** condition **then**
sequence_of_statements }
[**else**
sequence_of_statements]
end if [*if*_label] ;

<u>incomplete_type_declaration</u> ::= **type** identifier ;

index_constraint ::= (discrete_range { ~~, discrete_range~~ })

index_specification ::=
discrete_range
| *static*_expression

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

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

instantiated_unit ::=
[**component**] *component*_name
| **entity** *entity*_name [(*architecture*_name)]
| **configuration** *configuration*_name

instantiation_list ::=
*instantiation*_label {, *instantiation*_label}
| **others**
| **all**

integer ::= digit { [underline] digit }

integer_type_definition ::= range_constraint

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

interface_declaration ::=
    interface_constant_declaration
  | interface_signal_declaration
  | interface_variable_declaration
  | ~~interface_file_declaration~~

interface_element ::= interface_declaration

~~interface_file_declaration~~ ::=
  file identifier_list : subtype_indication

interface_list ::=
  interface_element {; interface_element}

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

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

iteration_scheme ::=
    ~~**while** condition~~
  | **for** *loop*_parameter_specification

label ::= identifier

letter ::= upper_case_letter | lower_case_letter

letter_or_digit ::= letter | digit

library_clause ::= **library** logical_name_list ;

library_unit ::=
    primary_unit
  | secondary_unit

literal ::=
    numeric_literal
  | enumeration_literal
  | string_literal
  | bit_string_literal
  | ~~**null**~~

logical_name ::= identifier

logical_name_list ::= logical_name { , logical_name }

logical_operator ::= **and** | **or** | **nand** | **nor** | **xor** | **xnor**

loop_statement ::=
  [*loop*_label:]
    [iteration_scheme] **loop**
      sequence_of_statements
    end loop [*loop*_label] ;

miscellaneous_operator ::= ****** | **abs** | **not**

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

multiplying_operator ::= ***** | **/** | **mod** | **rem**

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

next_statement ::=
  [label:] **next** [*loop*_label] [**when** condition] ;

null_statement ::=
  [label:] **null** ;

numeric_literal ::=
    abstract_literal
  | ~~physical_literal~~

object_declaration ::=
    constant_declaration
  | signal_declaration
  | variable_declaration
  | <u>file_declaration</u>

operator_symbol ::= string_literal

options ::= [**guarded**] [<u>delay_mechanism</u>]

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

package_body_declarative_item ::=
subprogram_declaration
| subprogram_body
| type_declaration
| subtype_declaration
| constant_declaration
| ~~*shared*_variable_declaration~~
| <u>file_declaration</u>
| alias_declaration
| use_clause
| ~~group_template_declaration~~
| ~~group_declaration~~

package_body_declarative_part ::=
{ package_body_declarative_item }

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

package_declarative_item ::=
subprogram_declaration
| type_declaration
| subtype_declaration
| constant_declaration
| signal_declaration
| ~~*shared*_variable_declaration~~
| <u>file_declaration</u>
| alias_declaration
| component_declaration
| attribute_declaration
| attribute_specification
| ~~disconnection_specification~~
| use_clause
| ~~group_template_declaration~~
| ~~group_declaration~~

package_declarative_part ::=
{ package_declarative_item }

parameter_specification ::=
identifier **in** discrete_range

~~physical_literal~~ ::= [abstract_literal] *unit*_name

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

port_clause ::=
  port(port_list);

port_list ::= *port*_interface_list

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

prefix ::=
    name
  | function_call

primary_unit_declaration ::= identifier ;

primary ::=
    name
  | literal
  | aggregate
  | function_call
  | qualified_expression
  | type_conversion
  | ~~allocator~~
  | (expression)

primary_unit ::=
    entity_declaration
  | configuration_declaration
  | package_declaration

procedure_call ::= *procedure*_name [(actual_parameter_part)]

procedure_call_statement ::= [label:] procedure_call ;

process_declarative_item ::=
    subprogram_declaration
  | subprogram_body
  | type_declaration
  | subtype_declaration
  | constant_declaration
  | variable_declaration
  | file_declaration
  | alias_declaration
  | attribute_declaration
  | attribute_specification
  | use_clause
  | ~~group_template_declaration~~
  | ~~group_declaration~~

process_declarative_part ::=
  { process_declarative_item }

process_statement ::=
[*process*_label:]
[~~**postponed**~~] **process** [(sensitivity_list)] [**is**]
process_declarative_part
begin
process_statement_part
end [~~**postponed**~~] **process** [*process*_label] ;

process_statement_part ::=
{ sequential_statement }

~~protected_type_body~~ ::=
protected body
protected_type_body_declarative_part
end protected body [*protected_type*_simple_name]

~~protected_type_body_declarative_item~~ ::=
subprogram_declaration
| subprogram_body
| type_declaration
| subtype_declaration
| constant_declaration
| variable_declaration
| <u>file_declaration</u>
| alias_declaration
| attribute_declaration
| attribute_specification
| use_clause
| ~~group_template_declaration~~
| ~~group_declaration~~

~~protected_type_body_declarative_part~~ ::=
{ protected_type_body_declarative_item }

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

~~protected_type_declarative_item~~ ::=
subprogram_declaration
| attribute_specification
| use_clause

~~protected_type_declarative_part~~ ::=
{ protected_type_declarative_item }

~~protected_type_definition~~ ::=
protected_type_declaration
| protected_type_body