

Electronic system specification languages — Standard method for building VHDL models of component libraries

ICS 35.240.50

National foreword

This Published Document reproduces verbatim CENELEC Report R217-021:2001.

The UK participation in its preparation was entrusted to Technical Committee GEL/93, Design automation, which has the responsibility to:

- aid enquirers to understand the text;
- present to the responsible international/European committee any enquiries on the interpretation, or proposals for change, and keep the UK interests informed;
- monitor related international and European developments and promulgate them in the UK.

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

Cross-references

The British Standards which implement international publications referred to in this document may be found in the BSI Standards Catalogue under the section entitled “International Standards Correspondence Index”, or by using the “Find” facility of the BSI Standards Electronic Catalogue.

This Published Document, having been prepared under the direction of the Electrotechnical Sector Policy and Strategy Committee, was published under the authority of the Standards Policy and Strategy Committee on 13 March 2002

Summary of pages

This document comprises a front cover, an inside front cover, the CENELEC Report title page, pages 2 to 49 and a back cover.

The BSI copyright date displayed in this document indicates when the document was last issued.

Amendments issued since publication

Amd. No.	Date	Comments

© BSI 13 March 2002

ISBN 0 580 39096 9

English version

**Electronic system specification languages –
Standard method for building VHDL models
of component libraries**

This CENELEC Report has been prepared by TC 217, Electronic Design Automation (EDA). It was approved by TC 217 on 1997-07-07 and endorsed by the CENELEC Technical Board on 2001-03-06.

CENELEC members are the national electrotechnical committees of Austria, Belgium, Czech Republic, Denmark, Finland, France, Germany, Greece, Iceland, Ireland, Italy, Luxembourg, Malta, Netherlands, Norway, Portugal, Spain, Sweden, Switzerland and United Kingdom.

CENELEC

European Committee for Electrotechnical Standardization
Comité Européen de Normalisation Electrotechnique
Europäisches Komitee für Elektrotechnische Normung

Central Secretariat: rue de Stassart 35, B - 1050 Brussels

© 2001 CENELEC - All rights of exploitation in any form and by any means reserved worldwide for CENELEC members.

Ref. No. R217-021:2001 E

 **Documents**
Document Sharing Hub

Foreword

This CENELEC Report has been prepared by the Technical Committee CENELEC TC 217, Electronic Design Automation (EDA). Having been approved by TC 217 in July 1997, the need for its publication was confirmed on 2000-10-16 and endorsed by the CENELEC Technical Board on 2001-03-06.

The report is derived from two sources:

- 1) Standardisation within OMI (Open Microprocessor Initiative)
Esprit project 7267 which resulted in OMI-326 VHDL Coding Standard;
- 2) Lucent Technologies - Bell Labs Innovations internal standardisation efforts.

This report contains a high level of input from experienced VHDL model writers and valuable feedback from many users out of different sites all over the world.

Contacts:

<u>Name</u>	<u>Email</u>
Hans Sahm (Chair)	hsam@lucent.com
Claus Mayer	clausmayer@lucent.com
Jörg Pleickhardt	jpleickhardt@lucent.com
Pierre Ragon	pragon@lucent.com
Thomas Ziselsberger	tziselsberger@lucent.com
Andreas Preiss	apreiss@lucent.com

Controlling department: VHDL Steering Group

Version: A-1-0

Author(s): Lucent Technologies, Nürnberg

Contents

1	General principles	4
1.1	Introduction	4
1.2	Recommended reading	4
1.3	File management	5
2	Code layout	6
2.1	Naming	6
2.2	Commenting	8
2.3	File layout	10
2.4	Statement layout	13
3	Language usage	22
3.1	General usage	22
3.1.1	Data types	22
3.1.2	Packages	25
3.1.3	Configurations	25
3.1.4	Constants	25
3.1.5	Assertions	26
3.1.6	Qualified expressions and type conversion	26
3.1.7	Interface lists	26
3.1.8	Wait statements	27
3.1.9	Loops	27
3.2	Behavioral usage	28
3.2.1	General aspects of a behavioral model	28
3.2.2	Data types	29
3.2.3	Logic data types	29
3.2.4	Naming conventions	31
3.2.5	Event and signal handling	31
3.2.6	Subprograms	34
3.2.7	Variables	34
3.2.8	Deferred constants	34
3.2.9	Component binding	35
3.2.10	Initialization and default values	35
3.2.11	Miscellaneous	35
3.3	RTL usage	36
3.3.1	General aspects of the RTL model	36
3.3.2	Data types	38
3.3.3	VHDL objects and structural elements	40
3.3.4	Process structures	43
3.3.5	Subprograms	45
3.3.6	Architecture structures	46
3.3.7	Configurations	46
3.3.8	Loops	46
3.3.9	Naming conventions	47
	Annex A: Standard packages	48
	Annex B: References	49

1 General principles

1.1 Introduction

VHDL as a hardware description language offers almost complete freedom to the designer as how to model the design. While this gives a lot of power and flexibility, it also leads to problems as different modelling styles may restrict or even prevent exchangeability and reusability of models.

The purpose of the following chapters is to provide *rules* and *guidelines* for hand-coded VHDL models, with the definitions:

Rule: Anything is classified as rule which does significantly contribute to

- 1) *early detection of errors*,
- 2) *portability*,
- 3) *code efficiency*,
- 4) *code quality*,
- 5) *maintainability*,
- 6) *reusability*,
- 7) *readability*.

Guideline: Best common practice based on experience that would improve the synthesis or simulation for novice designers and hints for VHDL usage are classified as guideline.

All rules and guidelines in this document can be identified by a six digit symbolic name. This symbolic code can be used for references from supporting tools or companion documents. For an optical distinction rules are set in a bold face (e.g. [**CNAENG**]), while guidelines are set in an italic font (e.g. [*CNALEN*]). The first letter of the code refers to the main section to which a rule or recommendation belongs:

Section	Symbolic names
<u>File Management</u>	[F.]
<u>Code Layout</u>	[C.]
<u>General Usage</u>	[G.]
<u>Behavioral Usage</u>	[B.]
<u>RTL Usage</u>	[R.]

The second and third letters refer to subsections, while the remaining characters identify individual rules and guidelines.

Care has been taken, to keep this document as independent as possible from any particular VHDL implementation or VHDL tool. However, as indicated in the RTL Usage section, because of the different capabilities of VHDL synthesis tools, this section cannot be strictly tool independent.

1.2 Recommended reading

Usually, you don't have to work through this entire document in order to write VHDL code compliant with all the rules and guidelines. The entire file layout and statement layout sections are implicitly covered by the VHDL formatter vfmt [\[13\]](#), so you may decide to rely on automated formatting compliant to these sections.

Best approach for starting to read is with section General usage followed by section Behavioral usage for creating testbench or simulation models or by section RTL usage for creating synthesizable VHDL models. A code compliance checker like vlint [\[13\]](#) may be used for automatic rule compliance checks of VHDL sources for simulation and synthesis mode.

1.3 File management

Choosing an appropriate data structure for storage and retrieval of project data is an essential prerequisite for successful project management. For the mapping of primary and secondary design units to VHDL files, two alternatives are presented together with their advantages:

[FILMOD] Design units are organized as modules, which are sets of related design units: either an entity with its architectures and configurations or a package declaration together with its package body. The name of such a module must be either the name of the entity or the name of the package. Each module is kept in a separate file, which takes the name of the module together with the standard extension .vhdl or .vhd.

Main advantages of this approach are: related information is close together within one file and when several revisions of design units exist, an automatic relation of the revisions of corresponding entity, architecture and configuration exists.

File: *entityname.vhd(l)*

```
ENTITY entityname IS
END entityname;

ARCHITECTURE rtl OF
entityname IS
BEGIN

END rtl;
-- more architectures,
if any

CONFIGURATION conf1 OF
entityname IS

END conf1;
-- more configurations,
if any
```

File: *packagename.vhd(l)*

```
PACKAGE packagename IS
END packagename;

PACKAGE BODY
packagename IS

END packagename;
```

Figure 1 - Related design units are combined to modules

[FILGAT] Alternative choice is to put each design unit within a separate file with the proposed naming conventions:

ENTITY:	<entity>.vhd(l)
ARCHITECTURE:	<entity>-<architecture>.vhd(l)
PACKAGE:	<package>.vhd(l)
PACKAGE BODY:	<package>-body.vhd(l)
CONFIGURATION:	<configuration>-cfg.vhd(l) or <entity>-<configuration>.vhd(l)

Main advantage of this approach are less required compile runs since each design unit can be separately analysed without affecting other design units.

- [FILRCS]** All VHDL source files must be under control of a Revision Control System to ensure access to arbitrary versions.

For example, RCS could be used for revision control of VHDL sources.

- [FILTEX]** Even if a VHDL model has been derived from a code generator, for model exchanges, the VHDL description serves as a reference. The designer is responsible for maintaining consistency between graphical input and the corresponding VHDL representation.

Please note that tool support for RCS based revision control is fully documented in reference [\[13\]](#).

2 Code layout

2.1 Naming

The choice of appropriate and meaningful names strongly determines the quality of the created models, thus much effort should be spent to choose suitable object names. This section lists rules which apply to the selection of identifiers.

- [CNAENG]** All user-defined names must be chosen from the English language.
This rule also applies for models which are not supposed to be shared with other designers because the usage of national languages would restrict later exchangeability.

- [CNAKEY]** VHDL keywords (e.g. `PROCESS`) and predefined attribute names (e.g. `'LENGTH`) must be written using a consistent casing (either all uppercase or all lowercase letters). For user-defined identifiers, an uniform casing opposite to the casing of VHDL keywords must be applied.
You can use the configuration file of a formatter like `vfmt` [\[13\]](#) to change the character casing of source files consistently.

- [CNAVHD]** New keywords introduced in VHDL-93 must not be used as identifiers in VHDL-87 models as this reduces model compatibility and reusability.

- [CNAUND]** Within identifiers, use underscores to separate words and improve readability.

Example: use `buffer_is_empty` rather than `bufferisempty`.

- [CNAUNI]** Identifiers must be unique within the first 15 characters.
The VHDL standard does not impose restrictions on the length of identifiers other than that it must fit on one line, but some VHDL implementations may restrict the length of names. The purpose of this rule is to assure maximum portability across most VHDL implementations.

- [CNALEN]** The length of identifiers should be in the range of 3 to 30 characters.
This is a trade-off between the requirements of documentation and practical operation. Short names are usually less descriptive whereas excessive long names are harder to type and lead to more line wraps, which in turn reduce readability.

- [CNASCO] Choose the length of names according to the scope of their usage.
- While names used across a wide scope must be longer and more explicit, identifiers with narrow scope (e.g. in a `GENERATE` statement) may be quite short. However, even if you have narrow scope, choosing descriptive names may make your models more readable (e.g., use `row` and `column` instead of `i` and `j` for RAM addressing).
- [CNAFIL] The length of filenames (excluding possible path prefixes) must not exceed 28 characters.
- This is to avoid conflicts with restrictions within computer infrastructure, e.g. tar archive and NFS limitations.
- Because source filenames are derived from either entity or package names, this also affects the choice of entity and package identifiers.
- [CNAPRF] Use short prefixes or suffixes to indicate functional grouping of functions or objects. Separate prefixes and suffixes by an underscore from the rest of the name. If such a naming scheme has been set up, it must be well documented and used consistently.
- Example: you could use the prefix `io_` for all functions in an I/O subsystem.
- [CNASTA] Start identifiers with the most specific or important part and put generic or less important parts towards the end of the name.
- Example: use `x_coordinate` and `y_coordinate` rather than `coordinate_x` and `coordinate_y`.
- [CNAABR] Make names shorter by using a set of commonly accepted abbreviations for frequently used items. Such a list of abbreviations can be defined on a per-project basis and should be well documented. Once such a list has been established, the abbreviations must be used consistently, i.e. it is not allowed to use different abbreviations for the same item.

pointer	-> ptr	header	-> hdr
frame	-> frm	clock	-> clk
initialize	-> init	...	

Figure 2 - Example of an abbreviation list

- [CNAGEN] Avoid usage of generic names like `count` or `pointer`. Use specific names instead, which indicate, what this particular object is for or what it does.
- [CNALOW] Use a consistent naming scheme to denote active low signals (e.g. append suffix `_n`).
- [CNAENT] The name of an entity may be arbitrarily chosen according to the above rules. (Refer to sections "Architecture Names" and "Architecture Names RTL" for naming conventions for architectures on behavioral and on RT level.)

- [CNACFG] The name of the first (default) configuration is *entityname_conf*.
If an entity has more than one configuration, all remaining configurations are called *entityname_architecturename*.
Using a default configuration name reduces the number of changes in higher level instantiations, when a configuration changes.
- [CNACMP] In component instantiations, the component name should be equal to the name of the instantiated entity.
- [CNAPKG] Packages should have the suffix ' *_pkg* ' appended to their name.
Module specific packages should be named *modulename_pkg*.
Project specific packages should be named *projectname_contents_pkg*.
- [CNALAB] Every process must have a meaningful name (process label). Automatically generated (and therefore unreadable) names will be used as a default if no explicit name is supplied.
- [CNAINS] For instances, use meaningful names, similar to the entity name if this is appropriate.
Do not use simple instance names like "u1" (this might be acceptable if a graphical schematic information exists).
- [CNAILE] Instance names should not be excessively long because these names are used to describe the path within the debugger and long names lead to unreadable pathnames.
- [CNASTD] Names of predefined standard types or subtypes may not be used as object names or for own type declarations.
- [CNASUB] Names of predefined standard subprograms may not be used for own subprogram declarations.
- [CNAEXT] VHDL source files must have the standard filename extensions *.vhdl* or *.vhd* and must not contain uppercase letters. Within one project, only one type of extension should be used.
- [CNAPRO] Project teams may choose to use additional (file) naming conventions (prefixes/suffixes) provided that these conventions do not conflict with the conventions of this document.

2.2 Commenting

- [CCOENG] All comments must be written using the English language.
- [CCOLEN] Comments should contribute to a better understanding of a model and should neither be too short nor excessively long.
In general, well-chosen identifiers and a (to a certain extent) self-documenting codestyle are preferable to a talkative commenting.
You should keep in mind that a model will be written only once but will be read many more times and that a potential reader of the model does not necessarily have the same knowledge as the writer of the model.

- [CCOBLO]** Use the predefined heading comments shown in [section 2.3](#) for describing building blocks like `ENTITYs`, `ARCHITECTURES` or subprograms. Use one- or multiline block comments to delimit and describe logical blocks of code. Trailing comments describe single lines and should only be used to emphasize special situations.

```

----- short header -----
-- this is a multiline block comment with a header line.
-- The body of the comment contains a discription of the
-- block of code following the comment.
-- The purpose of the optional header line is to provide
-- a fast overview.
-----

----- one-line block header used as delimiter -----

frm_cnt <= assign_rec((OTHERS => 0)); -- trailing comment

```

Figure 3 - Examples of different comment styles

- [CCOSEP]** Clearly separate code and comments visually using blank lines and vertical alignment of trailing comments.

```

Example:
-- first example
out_sig   <= compare(in_sig); -- comment1
reset_all; -- comment2

-- second example (with separation and alignment)

out_sig   <= compare(in_sig);    -- vertical alignment
reset_all;                      -- of trailing comments

```

- [CCOADD]** Comments should not simply restate what is written in the code but should add real value.
- [CCOVAR]** Variants of a model should only be temporarily commented out for debugging purposes. To keep different variants permanently, they should be held in multiple architectures of an entity.
- [CCODOC]** Only fully documented sourcefiles may be checked in into the project database. In particular, it is not allowed to check in an uncommented version and to check in the commented source as a separate revision with equal functionality.
- [CCOMOD]** When modifying an existing model, existing comments must be immediately adapted appropriately.
- [CCOIND]** Comments must have the same indentation level as the code to which they refer.
- [CCOLAB]** All processes must be given a meaningful label for better documentation and easy addressing in sourcecode debuggers and synthesis tools. This label must be repeated in the corresponding `END PROCESS` statement.

[CCOEND] In ENTITY-, ARCHITECTURE, CONFIGURATION-, PACKAGE (BODY)- and subprogram declarations, the object name must be repeated in the corresponding END- statement.

[CCONES] When using nested structures with possibly long blocks of code, matching END statements should be commented with the starting condition.

Example:

```

IF level > 5 THEN
    IF ptr_set = 15 THEN

        -- lots of code here

    END IF ; -- ptr_set = 15
END IF ; -- level > 5

```

2.3 File layout

This section comprises rules for composing VHDL source files from separate design units. It also contains a set of standard headers designed for uniform documentation style.

[CFLGRP] Use empty lines to form logical groups of statements. Often, it makes sense to precede such a group of statements with a block comment.

Develop a scheme for separating design units, subprograms, etc. with blank lines and use it consistently throughout your models.

Example:

```

---- action one -----
-- this block comment describes the first block --
-- of statements --
-----

statement;
statement;

---- action two -----
-- this block comment describes the second block --
-- of statements --
-----

statement;

```

[CFLREL] Only related design units (Entity+Architecture+Configuration or Package+Package Body) may be combined into source files as shown in [Figure 1](#).

The order in which design units may appear in the source file is determined by the VHDL language, as any 'primary unit must be analysed prior to the analysis of any corresponding secondary unit' (LRM § 11.4).

[CFLLIN] The length of a source line may not exceed 80 characters.

[CFLFIL] It is recommended that a source file should not exceed 1 000 lines (about 15 to 20 pages of code).

[CFLCFF] Use formfeeds within comments (`--^L`) in the VHDL source code to separate design units visually and to structure source files.

FormFeeds should be placed within comments to improve portability because some VHDL compilers do not accept FormFeeds as whitespaces.

- [CFLHEA] It is recommended to use the standard headers shown in [Figures 5 to 10](#) to describe individual design units (entities, architectures, configurations and packages).
- [CFLMOR] If more than one architecture or configuration exist for an entity, each individual architecture and configuration must be preceded and documented by its own header.
- [CFLFHD] Each source file must start with a standard file header similar to the one shown in [Figure 4](#). Apart from information about the file itself, its author and a description of the file contents, it contains a standard copyright message.
 Testbench files or, input- or output files which are closely related to the source file or of which the filenames are hardcoded in the model must be listed in the 'Related Files' entry.
 The author field should not only contain the name(s) of the author(s) but also the name of the department and the phone extension or email address.
 The value of the project field should be set consistently among all project members to the full project name.
 The 'Simulator' and 'Synthesis' entries should be used to list the names and version numbers of the simulators and/or synthesis tools on which the model has been tested with.
 Problems with certain tools should also be mentioned there.
 The contents field should state the design units the file contains. Explicit descriptions of the design units should be given in the respective design unit headers.

```

-----
--
-- File           :
-- Related Files  :
--
-- Author(s)      :
-- Organization    :
--
-- Project        :
--
-- CreationDate   :
--
-- Simulator      :
-- Synthesis      :
--
-- Contents       :
--
-----
--                               Copyright
-----
--
--      Copyright (C) Lucent Technologies
--      Lucent Technologies - PROPRIETARY
--
-- Disclosure to third parties or reproduction in any form what-
-- soever, without prior written consent, is strictly forbidden
--
-----
--                               History
-----

```

Figure 4 - Standard file header for VHDL source files

[*CFLINC*] Entries in header files can be automatically filled with the appropriate values during header file inclusion from text editors or the vfmt formatter.

The values for the header fields can be provided as environment variables and should be set on a per-user basis (Author, Project) and system-wide (Organization, SimulatorId, SynthesisId). (See [13] for details on header file inclusion.)

```

-----
--
-- Entity      :
--
-- Author(s)   :
-- Organization :
--
-- Project     :
--
-- CreationDate :
--
-- Description  :
--
-----

```

Figure 5 - Standard header for VHDL entity declarations

```

-----
--
-- Architecture :
-- of Entity    :
--
-- Author(s)    :
-- Organization  :
--
-- Project      :
--
-- CreationDate :
--
-- Description   :
--
-----

```

Figure 6 - Standard header for VHDL architecture declarations

```

-----
--
-- Package      :
--
-- Author(s)    :
-- Organization  :
--
-- Project      :
--
-- CreationDate :
--
-- Description   :
--
-----

```

Figure 7 - Standard header for VHDL package declarations or package bodies

- [CFLAMO]** The amount of description which is necessary for a process or a subprogram depends on the complexity and size of the process and subprograms.
If a procedure or process modifies its environment (global signals or variables defined outside a procedure), this must be explicitly documented as 'Side Effects'.

```

-----
--
-- Subprogram      :
--
-- Description     :
--
-- Side Effects    :
--
-----

```

Figure 8 - Standard header for VHDL subprograms

```

-----
--
-- Process         :
--
-- Description     :
--
-- Side Effects    :
--
-----

```

Figure 9 - Standard header for VHDL processes

2.4 Statement layout

Apart from a few general principles how VHDL statements should be coded, the vast majority of this section consists of templates for the most important language constructs in a semiformal representation.

- [CSLONE]** Only one statement may appear on a line.

While this leads to longer source files, it greatly improves readability and facilitates stepping through the code with a source level debugger.

- [CSLBLK]** Where appropriate, write your code in a block structure with the enclosing keywords or parenthesis aligned to the current indentation level and the enclosed blocks indented by one tabstop.

Examples:

```

ARCHITECTURE rtl OF ent IS
  -- declaration section indented by 1 tabstop
BEGIN
  -- statement section indented by 1 tabstop
END rtl;

PORT (
  -- port declarations indented by 1 tabstop
  clock : IN      std_ulogic;
  q1_out: OUT     std_ulogic
);

```

[CSLIND] An indentation depth of 4 columns must be used across all used tools. Indentation depth of 2 or 3 columns are however also acceptable. Refer to documentation [13] for hints how to customize a number of tools to comply with this requirement.

[CSLVER] Use vertical alignment for signal and variable assignments and declarations to achieve a readable layout.

Example:

```
reset_all <= '0';
qn_sel <= NOT q_in;

reset_all <= '0';
qn_sel      <= NOT q_in;
```

[CSLWRP] Use a consistent scheme for wrapping long statements, which do not fit on a single line in a file. The indentation of continuation lines must be clearly distinguishable from the current indentation level.

[CSLSPA] Selected names, record and attribute expressions are closely related references to a single object and must be written without spaces. No spaces should also be put between an array name and the array index and a subprogram name and its parameter list.

Examples:

```
std.textio.ALL
complex_number.real_part
str'LENGTH
data_bus(5 DOWNT0 0)
conv_unsigned(int_val, size)
```

[CSLPAR] Explicitly parenthesize logical expressions so that no reference to the precedence rules are necessary to figure out the meaning of the expressions. Since precedence rules can vary from one language to another, parenthesis may help avoid potential ambiguities.

[CSLUNY] No spaces should be written between the unary '+' and '-' arithmetic operators and their operands. The same number of spaces (usually one) should be put around all binary operators.

Examples:

```
a_mult := -1 * b_in;
c_calc <= a_in + b_in - c_in;
```

[CSLPHY] Physical literals must be written with value and the unit of measurement separated by whitespaces.

Examples:

```
10 ns, 30 mm
```

[CSLLET] Letters occurring in numerical literals such as the exponentiation character, the character indicating the base of a based literal or hexadecimal literals must be written in lowercase.