

IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society

Approved 16 September 1999

IEEE-SA Standards Board

Abstract: A standard syntax and semantics for VHDL register transfer level (RTL) synthesis is defined. The subset of IEEE 1076 (VHDL) that is suitable for RTL synthesis is defined, along with the semantics of that subset for the synthesis domain.

Keywords: pragma, register transfer level (RTL), synthesis, VHDL

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

Copyright © 2000 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 9 March 2000. Printed in the United States of America.

Print: ISBN 0-7381-1819-2 SH94792
PDF: ISBN 0-7381-1820-6 SS94792

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

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

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

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.

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 all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

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.

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

Introduction

[This introduction is not part of IEEE Std 1076.6-1999, IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis.]

This standard describes a syntax and semantics for VHDL RTL synthesis. It defines the subset of IEEE Std 1076-1993 (VHDL) that is suitable for RTL synthesis as well as the semantics of that subset for the synthesis domain. This standard is based on IEEE Std 1076-1993, 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 1076-1993. 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 electronics engineers.

Work on this standard was initiated by the 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 subsets donated by a number of companies and groups.

After the PTAB approved Draft 1.5 with an overwhelming affirmative response, an IEEE project authorization request (PAR) was obtained for IEEE standardization. Most of the members of the original Working Group continued to be part of the Pilot Group of P1076.6 that led the technical work.

Participants

At the time this standard was balloted, the 1076.6 Pilot Team consisted of the following individuals:

Jayaram Bhasker, *Chair*

Rob Anderson, *Compiler Directives*

David Bishop, *Web and Reflector Administrator*

Apurva Kalia, *Semantics Task Leader*

Douglas J. Smith, *Editor/Attributes Task Leader*

Lance G. Thompson, *Syntax Task Leader*

Victor Berman
Dominique Borrione
Dennis B. Brophy
Ben Cohen
Colin Dente
Wolfgang Ecker

Robert A. Flatt
Christopher Grim
Rich Hatcher
Masamichi Kawarabayashi
Jim Lewis
Sanjiv Narayan

Doug Perry
Steven E. Shultz
Fur-Shing Tsai
Jim Vellenga
Eugenio Villar
Nels Vander Zanden

Many individuals from different organizations contributed to the development of this standard. In particular, the following individuals contributed to its development by being part of the Synthesis Interoperability 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 the development of this standard.

The following members of the balloting committee voted on this standard:

Mostapha Aboulhamid	Brian S. Griffin	Gabe Moretti
Phillip R. Acuff	Andrew Guyler	David S. Morris
Bill Anker	Pauline C. Haddow	Gerald Musgrave
Peter J. Ashenden	Michael J. Haney	Wolfgang W. Nebel
Jesus Avila-Casa	William A. Hanna	Kevin O'Brien
Stephen A. Bailey	Donald F. Hanson	Serafin Olcoz
Pete Bakowski	M. M. Kamal Hashmi	William R. Paulsen
David L. Barton	Rich Hatcher	Paolo Prinetto
Victor Berman	Jim Heaton	Johan Sandstrom
Jayaram Bhasker	Geir Hedemark	Hisashi Sasaki
William Billowitch	John Hillawi	Quentin G. Schmierer
Robert Blackburn	John Hines	Steven E. Schultz
Martin J. Bolton	Pradip Jha	Toru Shonai
Dominique Borrione	Apurva Kalia	Douglas J. Smith
Dennis B. Brophy	Takashi Kambe	Joseph J. Stanco
Patrick K. Bryant	Osamu Karatsu	Atsushi Takahara
Larrie Carr	Jake Karrfalt	Thomas D. Tessier
Shir-Shen Chang	Masamichi Kwarabayashi	Lance G. Thompson
Luc Claesen	Steven Kelem	Paul Traynar
Edmond S. Cooley	Robert H. Klenke	Yatin K. Trivedi
Alan Coppola	Venkatram Krishnaswamy	Cary Ussery
Ronnie W. Craig	Vello Kukuk	Radha Vaidyanathan
David C. Crohn	Gunther Lehmann	Ranganadha R. Vemuri
Clifford E. Cummings	Jim Lewis	Eugenio Villar
Timothy R. Davis	Victor Martin	Ronald Waxman
Steven L. Drager	Peter Marwedel	J. Richard Weger
Douglas D. Dunlop	John McCluskey	Ron Werner
George E. Econonuakos	Michael D. McKinney	John Williams
Robert A. Flatt	Paul J. Menchini	William R. Young
Scott Flinchbaugh	Jean P. Mermet	Tetsuo Yutani
Rita A. Glover	Egbert Molenkamp	Jan Zegers
Kenji Goto	Jaun Moreno	Mark Zwolinski

When the IEEE-SA Standards Board approved this standard on 16 September 1999, it had the following membership:

Richard J. Holleman, *Chair*
Donald N. Heirman, *Vice Chair*
Judith Gorman, *Secretary*

Satish K. Aggarwal	James H. Gurney	Louis-François Pau
Dennis Bodson	Lowell G. Johnson	Ronald C. Petersen
Mark D. Bowman	Robert J. Kennelly	Gerald H. Peterson
James T. Carlo	E. G. "Al" Kiener	John B. Posey
Gary R. Engmann	Joseph L. Koepfinger*	Gary S. Robinson
Harold E. Epstein	L. Bruce McClung	Akio Tojo
Jay Forster*	Daleep C. Mohla	Hans E. Weinrich
Ruben D. Garzon	Robert F. Munzner	Donald W. Zipse

*Member Emeritus

Also included is the following nonvoting IEEE-SA Standards Board liaison:

Robert E. Hebner

Janet Rutigliano
IEEE Standards Project Editor

Contents

1.	Overview	1
1.1	Scope	1
1.2	Compliance to this standard	1
1.3	Terminology	2
1.4	Conventions	2
2.	References	3
3.	Definitions	3
4.	Predefined types	4
5.	Verification methodology	5
5.1	Combinational verification	5
5.2	Sequential verification	6
6.	Modeling hardware elements	6
6.1	Edge-sensitive sequential logic	7
6.2	Level-sensitive sequential logic	11
6.3	Three-state and bus modeling	12
6.4	Modeling combinational logic	12
7.	Pragmas	12
7.1	Attributes	12
7.2	Metacomments	13
8.	Syntax	14
8.1	Design entities and configurations	14
8.2	Subprograms and packages	19
8.3	Types	23
8.4	Declarations	28
8.5	Specifications	34
8.6	Names	36
8.7	Expressions	38
8.8	Sequential statements	43
8.9	Concurrent statements	49
8.10	Scope and visibility	54
8.11	Design units and their analysis	55
8.12	Elaboration	56
8.13	Lexical elements	56
8.14	Predefined language environment	56
	Annex A (informative) Syntax summary	59

IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis

1. Overview

1.1 Scope

This standard defines a means of writing VHSIC hardware description language (VHDL) that guarantees the interoperability of VHDL descriptions among any register transfer level (RTL) synthesis tools that comply with this standard. Compliant synthesis tools may have features above those required by this standard. This standard defines how the semantics of VHDL shall be used; for example, to model level- 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.

The use of this standard should enhance the portability of VHDL designs across synthesis tools conforming to this standard. It should also minimize the potential for functional simulation mismatches between models both before 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 the tool

- a) Accepts all models that adhere to the model compliance definition 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 outlined in Clause 5

1.3 Terminology

The word *shall* indicates mandatory requirements to be followed strictly in order to conform to the standard and from which no deviation is permitted (*shall* equals *is required to*). The word *should* indicates 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) if it produces 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. 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.

1.4 Conventions

This standard uses the following conventions:

- a) VHDL reserved words (such as **downto**) are in boldface, and all other VHDL identifiers (such as REVERSE_RANGE or FOO) are in uppercase letters.
- 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 “>” 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 any such examples 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 with this standard (except as noted otherwise).

2. References

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

IEEE Std 1076-1993, IEEE Standard VHDL Language Reference Manual.¹

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

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

3. Definitions

Terms used within this standard, but not defined in this clause, are from IEEE Std 1076-1993², IEEE Std 1164-1993, and/or IEEE Std 1076.3-1997.

3.1 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.

3.2 don't care value: The enumeration literal ‘-’ of the type STD_ULOGIC (or subtype STD_LOGIC) defined by IEEE Std 1164-1993.

3.3 edge-sensitive storage element: A 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, and
- b) Preserves the last value propagated at all other times, except when any asynchronous control inputs become active.

(For example, a flip-flop.)

3.4 high-impedance value: The enumeration literal “Z” of the type STD_ULOGIC (or subtype STD_LOGIC) defined by IEEE Std 1164-1993.

(For example, a latch.)

3.5 Language Reference Manual (LRM): The IEEE VHDL Language Reference Manual (IEEE Std 1076-1993).

3.6 level-sensitive storage element: A 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, and
- b) Preserves the last value propagated at all other times, except when any asynchronous control inputs become active.

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

¹IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA (<http://www.standards.ieee.org/>).

²References are listed in Clause 2.

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

3.9 metalogical value: One of the enumeration literals, 'U', 'X', 'W', or '-', of the type STD_ULOGIC (or subtype STD_LOGIC) defined by IEEE Std 1164-1993.

3.10 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.

3.11 register transfer level (RTL): 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, 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.

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

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

3.14 vector: A one-dimensional array.

3.15 well-defined: Containing no metalogical or high-impedance element values.

3.16 synthesis-specific attribute: An attribute recognized by an RTL synthesis compliant tool as described in 7.1.

3.17 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.

4. Predefined types

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

- a) BIT, BOOLEAN, and BIT_VECTOR as defined in IEEE Std 1076-1993
- b) CHARACTER and STRING as defined in IEEE Std 1076-1993
- c) INTEGER as defined in IEEE Std 1076-1993
- d) STD_ULOGIC, STD_ULOGIC_VECTOR, STD_LOGIC, and STD_LOGIC_VECTOR as defined by the package STD_LOGIC_1164 in IEEE Std 1164-1993
- e) SIGNED and UNSIGNED as defined by the VHDL package NUMERIC_BIT in IEEE Std 1076.3-1997
- f) SIGNED and UNSIGNED as defined by the VHDL package NUMERIC_STD in IEEE Std 1076.3-1997

No array types, other than those listed in items (e) and (f) above, shall be used to represent signed and 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 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 models, 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. Since 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.

Input stimulus shall comply to the following criteria:

- a) Input data does not contain metalogical values.
- b) Input data may contain ‘H’ and ‘L’ on inputs, in which case they 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) After asynchronous set/reset signals go from active to inactive, there must be enough time to take care of setup/hold times of the sequential elements before clock and/or input data change.
- e) For edge-sensitive-based designs, primary inputs of the design must change far enough in advance for the edge-sensitive storage element input data to not violate the setup times with reference to the active clock edge. Also, the input data must remain stable for long enough to respect the hold times with respect to the active clock edge.
- f) For level-sensitive storage element based designs, primary inputs of the design must change far enough in advance for the level-sensitive storage element input data to respect the setup times. Also, the input data must remain stable for long enough to respect the hold times.

NOTE—A synthesis tool may define metalogical 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 shall be examined. To verify the combinational logic portion of a model, the following sequence of events shall be performed 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 to never 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 used for combinatorial verification shall be performed during sequential verification: change the inputs, compute the results, compare the outputs. For sequential verification, however, 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 performed just before the active clock edge and the non-clock inputs can change 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. In the absence of changes to the inputs of the level-sensitive model, if one or more internal values or outputs of the model never reach a steady state (oscillatory behavior), then it 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, and three-state drivers.

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 and may not generate the suggested hardware inferences, or it may generate a different set of hardware inferences. 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) with a minimum subset of '0' and '1'. 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 donut 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 function `RISING_EDGE` shall represent a rising edge and the function `FALLING_EDGE` shall represent a falling edge. `RISING_EDGE` and `FALLING_EDGE` are the functions declared by either the package `STD_LOGIC_1164` defined in IEEE Std 1164-1993, or by the package `NUMERIC_BIT` defined in IEEE Std 1076.3-1997.

```

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

```

6.1.2.1 Positive edge clock

The following expressions shall represent a positive clock edge when used as a condition in an **if** statement (positive <clock_edge>):

```

— RISING_EDGE( clk_signal_name )
— clk_signal_name'EVENT and clk_signal_name = '1'
— clk_signal_name = '1' and clk_signal_name'EVENT
— not clk_signal_name'STABLE and clk_signal_name = '1'
— clk_signal_name = '1' and not clk_signal_name'STABLE

```

The following expressions shall represent a positive clock edge when used as a condition in a **wait until** statement (positive <clock_edge> or <clock_level>):

```

— RISING_EDGE(clk_signal_name)
— clk_signal_name = '1'
— clk_signal_name'EVENT and clk_signal_name = '1'
— clk_signal_name = '1' and clk_signal_name'EVENT
— not clk_signal_name'STABLE and clk_signal_name = '1'
— clk_signal_name = '1' and not clk_signal_name'STABLE

```

6.1.2.2 Negative edge clock

The following expressions shall represent a negative clock edge when used as a condition in an **if** statement (negative <clock_edge>):

```

— FALLING_EDGE(clk_signal_name)
— clk_signal_name'EVENT and clk_signal_name = '0'
— clk_signal_name = '0' and clk_signal_name'EVENT
— not clk_signal_name'STABLE and clk_signal_name = '0'
— clk_signal_name = '0' and not clk_signal_name'STABLE

```

The following expressions shall represent a negative clock edge when used as a condition in a **wait until** statement (negative <clock_edge> or <clock_level>):

```

— FALLING_EDGE(clk_signal_name)
— clk_signal_name = '0'
— clk_signal_name'EVENT and clk_signal_name = '0'
— clk_signal_name = '0' and clk_signal_name'EVENT
— not clk_signal_name'STABLE and clk_signal_name = '0'
— clk_signal_name = '0' and not clk_signal_name'STABLE

```

6.1.3 Modeling edge-sensitive storage elements

A synchronous assignment takes place when a signal or variable is updated as a direct result of a clock edge expression evaluation to true.

A signal updated with a synchronous assignment should model one or more edge-sensitive storage elements.

A variable updated in a synchronous assignment should model an edge-sensitive storage element. If simulation semantics suggest that the value of the variable is read before it is written, then an edge-sensitive storage element should be modeled by the variable. By optimization, the generated edge-sensitive storage may be eliminated.

Only one clock edge shall be allowed per **process** statement (including any procedures called within the **process**). Conditional or selected signal assignments shall not be used to model an edge-sensitive storage element (see 8.9.5).

No **wait** statements are allowed in a procedure (8.2.2).

6.1.3.1 Using the “if” statement

An edge-sensitive storage element may be modeled using a clock edge with an **if** statement. The template for modeling such an edge-sensitive storage element shall be

```

[process_label:] process (<clock_signal>)
  <declarations>
begin
  if <clock_edge> then
    <sequence_of_statements>
  end if;
end process [process_label];

```

The clock signal in <clock_edge> shall be listed in the process sensitivity list.

Sequential statements preceding or succeeding the **if** statement shall not be supported.

Example:

```

DFF: process(CLOCK)
begin
  if CLOCK'EVENT and CLOCK = '1' then
    Q <= D; -- Q models a rising edge-triggered storage element
  end if;
end process DFF;

```

6.1.3.2 Using the “wait” statement

An edge-sensitive storage element may be modeled using a clock edge as a condition in a **wait until** statement. The **wait until** statement shall be the first statement in the **process**. No additional **wait until** statements shall appear within such a **process**, including any procedures called within the **process**. The template for modeling such an edge-sensitive storage element shall be

```

[process_label:]
process
  <declarations>
begin
  wait until <clock_edge>; -- this must be the first statement in the process
  <sequence_of_statements>
end process [process_label];

```

NOTES

1—Because the **wait until** statement must appear as the first statement of the process, an asynchronous override (set or reset) of edge-sensitive storage elements can not be represented using the **wait until** statement form.

2—Conditional or selected signal assignments shall not be used to represent edge-sensitive storage elements.

Example:

```

DFF1: process
begin
  wait until CLOCK = '0';
  Q <= D; -- Q models a falling edge-triggered storage element
end process DFF1;

```

Example:

```

DFF2: process
  variable VAR: UNSIGNED(3 downto 0);
begin
  wait until CLOCK = '1';
  VAR := VAR + 1;
  COUNT <= VAR;
end process DFF2;

```

```

-- Variable VAR should model four rising edge-triggered storage elements because the
-- value of VAR is read in the first assignment before its value is assigned.
-- By optimization, some edge-triggered storage elements may be eliminated.

```

Example:

```
DFF3: process
  variable VAR: UNSIGNED(3 downto 0);
begin
  wait until CLOCK = '1';
  VAR := COUNT;      -- Variable is written prior to being read.
  VAR := VAR + 1;     -- VAR is combinational.
  COUNT <= VAR;       -- Count models edge-sensitive storage elements.
end process DFF3;

-- Variable VAR should not model edge-sensitive storage elements because VAR is
-- assigned a value before its value is read.
```

6.1.3.3 With asynchronous control

A variable or a signal that is synchronously assigned may also be asynchronously assigned to model asynchronous set/reset edge-sensitive storage elements. Such a variable or a signal models an asynchronous set/reset edge-sensitive storage element. The template for representing such edge-sensitive storage elements shall be

```
[process_label:]
process (<clock_signal>, <asynchronous_signals>)
  <declarations>
begin
  if <condition1> then
    <sequence_of_statements>
  elsif <condition2> then
    <sequence_of_statements>
  elsif <condition3> then
    ...
  elsif <clock_edge> then
    <sequence_of_statements>
  end if;
end process [process_label];
```

The **if** branches preceding the last clock edge branch represents the asynchronous set/reset logic.

A clock edge shall only appear in the last **elsif** condition.

Sequential statements, as used in the template above, shall not include any **if** statements conditional on a clock edge.

The sensitivity list of the process shall include all of the following:

- The clock signal sensed by the clock edge expression
- All signals sensed by the remaining conditions of the **if** statement
- All signals sensed by the sequential statements governed by the remaining conditions of the **if** statement other than the clock edge expression

No signals other than those identified in the above list shall appear in the sensitivity list.

The order of the signals in the sensitivity list is not important.

Sequential statements preceding or succeeding the **if** statement shall not be supported.

NOTES

1—Asynchronous set-reset conditions are level sensitive; that is, they cannot contain a clock edge expression. Additionally, these conditions have a higher priority than the clock edge condition.

2—It is not necessary to describe both set and reset cases if the desired implementation does not require both of these features. Either or both may be modeled in the RTL model.

3—The VHDL semantics shall be followed in resolving any priority between set and reset.

Example:

```
AS_DFF: process (CLOCK, RESET, SET, SET_OR_RESET, A)
begin
    if RESET = '1' then
        Q <= '0';
    elsif SET = '1' then
        Q <= '1';
    elsif SET_OR_RESET = '1' then
        Q <= A;
    elsif CLOCK'EVENT and CLOCK = '1' then
        Q <= D;
    end if;
end process AS_DFF;

-- Signal Q models an asynchronous reset/set rising edge triggered
-- edge-sensitive storage element. The reset expression is RESET, the set
-- expression is SET, and SET_OR_RESET may be either a reset condition or a set
-- condition according to the value of A.
```

6.2 Level-sensitive sequential logic

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

- a) The signal (or variable) is assigned either directly in a process, or assigned within a subprogram invoked within the process, and the process contains no clock edge construct.
- b) There are executions of the process that do not execute an explicit assignment (via an assignment statement) to the signal (or variable).

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

- a) The signal (or variable) is assigned in a process that contains no clock edge construct.
- b) There are executions of the process in which the value of the signal (or variable) is read before its assignment.

The **process** sensitivity list shall contain all signals read within the **process** statement. Processes with incomplete sensitivity lists are not supported.

NOTES

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

2—Conditional or selected signal assignments shall not be used to model a level-sensitive storage element (see 8.9.5).

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

4—It is recommended to avoid a modeling style in which the value of a signal or variable is read before its assignment. This would avoid the generation of unwanted storage elements where none might be intended.

Example:

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

6.3 Three-state and bus modeling

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 assignment to “Z” shall be a conditional assignment; that is, assignment occurs under the control of a condition.

For a signal that has multiple drivers, if one driver has an assignment to “Z,” all drivers shall have at least one assignment to “Z.”

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), then a synthesis tool may infer sequential elements on all inputs of the three-state logic.

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

Concurrent signal assignment statements (see 8.9.5) and concurrent procedure calls (see 8.9.3) always model combinational logic.

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

7. Pragmas

Pragmas influence how a model is synthesized. The following pragmas may appear within the VHDL code:

- a) Attributes
- b) Metacomments

7.1 Attributes

Only one attribute with a synthesis-specific interpretation shall be supported for synthesis: ENUM_ENCODING. All others shall be ignored.

7.1.1 ENUM_ENCODING attribute

An attribute named ENUM_ENCODING shall provide a means of encoding enumeration type values. The attribute specification for this attribute shall specify the encoding of the enumeration type literals in the form of a string. This string 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 are to be ignored. All tokens shall be composed of the same number of characters (ignoring the underscore characters). The following declares an enumeration type and attribute ENUM_ENCODING:

```
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.

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

Token <token1> specifies the encoding for <enum_lit1>, <token2> specifies the encoding for <enum_lit2>, and so on.

This attribute shall only decorate an enumeration type.

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.
```

User-defined attribute declarations and their specifications shall be ignored.

7.2 Metacomments

Two metacomments provide for conditional synthesis control. They shall be

- RTL_SYNTHESIS OFF
- RTL_SYNTHESIS ON

A synthesis tool shall ignore any VHDL code after the “RTL_SYNTHESIS OFF” directive and before any subsequent “RTL_SYNTHESIS ON” directive.

Metacomments differing only in the use of corresponding uppercase and lowercase letters shall be considered the same.

The source code as a whole, including ignored constructs, shall conform to IEEE Std 1076-1993. The source code, exclusive of constructs ignored because of the metacomments, shall be compliant with 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

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

Ignored:

— Entity_statement_part

Not supported:

— Entity_declarative_part
— Reserved word **entity** after reserved word **end**

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
```

Types allowed in the generic interface list of the entity_header shall be those described in 8.4.3.2.

Supported:

- Generic_list

b) Ports

```
port_list ::= port_interface_list
```

Supported:

- Port_list

Ignored:

- Initial values in 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

```

Not supported:

- Entity_declarative_part
- Entity_declarative_item

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; therefore, it 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 corresponding to a given entity declaration

Not supported:

- Global signal interactions between architectures
- Reserved word **architecture** after reserved word **end**

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
- Alias_declaration
- Configuration_specification
- Disconnection_specification
- User-defined attribute declarations and their specifications, except as described in 7.1.

Not supported:

- Shared_variable_declaration
- Group_template_declaration
- Group_declaration

A **use** clause shall only reference the selected name of a package, which may, in turn, reference all of (or a particular item_name within) the package.

Attribute declarations and attribute specifications will be supported only for the synthesis-specific attributes described in 7.1. All other attribute declarations and attribute specifications shall be ignored.

8.1.2.2 Architecture statement part

```

architecture_statement_part ::=
{ concurrent_statement }

```

Supported:

- Architecture_statement_part

as defined 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

Not supported:

- Configuration_declarative_part
- Configuration_declarative_item
- Reserved word **configuration** after reserved word **end**

Configuration declaration shall be supported to the extent of specifying the architecture to be associated with the top-level entity of a synthesized design hierarchy.

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

Not supported:

- Use_clause
- Index_specification
- Configuration_item
- Block_statement_label
- Generate_statement_label

Use clause shall not be supported in this context.

Block specification shall only be an architecture name.

Configuration declaration shall only be used to select the architecture to be used with the top-level entity.