IEEE P1076.6/D2.01
Draft Standard For VHDL Register Transfer Level Synthesis

Prepared by the VHDL Synthesis Interoperability Working Group of the
Design Automation Standards Committee

IEEE Standards Activities Department
Standards Licensing and Contracts
445 Hoes Lane, P.O. Box 1331
Piscataway, NJ 08855-1331, USA

Introduction

(This introduction is not part of IEEE P1076.6, Draft Standard for VHDL Register Transfer Level Synthesis).

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

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 the IEEE 1076 standard. 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.

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.

Participants

At the time this standard was completed, the P1076.6 Pilot Team comprised of the following individuals:

Rob Anderson (Compiler directives)
Victor Berman
J. Bhasker (Working Group Chair)
David Bishop (Web and reflector admin)
Dominique Borrione
Denis Brophy
Ben Cohen
Colin Dente
Wolfgang Ecker
Bob Flatt
Christopher Grimm
Rich Hatcher
Apurva Kalia (Semantics task leader)
Masamichi Kawarabayashi
Jim Lewis
Sanjiv Narayan
Doug Perry
Steve Schultz
Doug Smith (Editor / Attibutes task leader)
Lance Thompson (Syntax task leader)
Fur-Shing Tsai
Jim Vellenga
Eugenio Villar
Nels Vander Zanden

Many individuals from different organizations contributed to the development of 1076.6. 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 email reflector also contributed to this development.
The following persons were on the ballotting committee that approved this document for submission to the Standards Board:

<list ballotting committee here>

This page is intentionally blank

Contents

1. Overview

1.1 Scope

This standard defines a means of writing VHDL that guarantees the interoperability of VHDL descriptions between any register transfer level 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.

Use of this standard should enhance the portability of VHDL designs across synthesis tools conforming to this standard. It should also minimize the potential of 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, and

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 section 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:

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) The body of the text of this standard uses boldface to denote VHDL reserved words (such as **downto**) and upper case to denote all other VHDL identifiers (such as REVERSE_RANGE or FOO).

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 this section 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).

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 1076-1993, IEEE Standard VHDL Language Reference Manual.

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

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

3. Definitions

Terms used within this standard but not defined in this section are assumed to be from IEEE Std 1076-1993 IEEE Std 1164-1993 and 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: Any edge-sensitive 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.

3.5 level-sensitive storage element: Any level-sensitive 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 latch.)

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

3.7 LRM: The IEEE VHDL language reference manual, that is, IEEE Std 1076-1993.

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 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, 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 register transfer level 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 Section 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 by 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 (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

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 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.

The input stimulus shall comply with the following criteria:

a) Input data does not contain metalogical 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 take care of the setup/hold times of the sequential elements in the design.

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 respect the setup times with respect 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 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, 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 section specifies styles for modeling hardware elements such as edge-sensitive storage elements, level-sensitive storage elements and three-state drivers.

This section 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 section 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 section 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 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 function RISING_EDGE shall represent a rising edge and the function FALLING_EDGE shall represent a falling edge, where RISING_EDGE and FALLING_EDGE are the functions declared either by the package STD_LOGIC_1164 of IEEE Std 1164-1993 or by the package NUMERIC_BIT defined by 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>):

a) `RISING_EDGE( clk_signal_name )`

b) `clk_signal_name'EVENT and clk_signal_name = '1'`

c) `clk_signal_name = '1' and clk_signal_name'EVENT`

d) `not clk_signal_name'STABLE and clk_signal_name = '1'`

e) `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>):

a) `RISING_EDGE(clk_signal_name)`

b) `clk_signal_name = '1'`

c) `clk_signal_name'EVENT and clk_signal_name = '1'`

d) `clk_signal_name = '1' and clk_signal_name'EVENT`

e) `not clk_signal_name'STABLE and clk_signal_name = '1'`

f) `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>):

a) `FALLING_EDGE(clk_signal_name)`

b) `clk_signal_name'EVENT and clk_signal_name = '0'`

c) `clk_signal_name = '0' and clk_signal_name'EVENT`

d) `not clk_signal_name'STABLE and clk_signal_name = '0'`

e) `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>):

a) `FALLING_EDGE(clk_signal_name)`

b) `clk_signal_name = '0'`

c) `clk_signal_name'EVENT and clk_signal_name = '0'`

d) `clk_signal_name = '0' and clk_signal_name'EVENT`

e) `not clk_signal_name'STABLE and clk_signal_name = '0'`

f) `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 **procedure**s called within the **process**). Conditional or selected signal assignments shall not be used to model a 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 **procedure**s 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];
```

Note 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.

Note 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:

a) The clock signal sensed by the clock edge expression.

b) All signals sensed by the remaining conditions of the **if** statement.

c) 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.

Note 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.

Note 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.

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

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

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

Note 3 -- When a signal is assigned from within a procedure it shall have the same inference semantics as a signal assignment from within a process.

Note 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 6.2 and 6.3, 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 the 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 (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). 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.

```
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:

a) -- RTL_SYNTHESIS OFF

b) -- 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 to the terms of this standard.

Note 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.

Note 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. 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 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, or a particular item_name within the package).

Attribute declarations and attribute specifications will be supported only for the synthesis-specific attributes described in Section 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 Concurrent statements.

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 only 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_expresion

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.

8.1.3.2 Component configuration

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

Not 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

Not supported:

- reserved words **pure** and **impure**

8.2.1.1 Formal parameters

```
formal_parameter_list ::= parameter_interface_list
```

Supported:

- formal_parameter_list

A subprogram shall not assign to an element or a slice of an unconstrained out parameter unless the corresponding actual parameter in each call of the subprogram is an identifier.

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_specification
- subprogram_declarative_part
- subprogram_declarative_item
- subprogram_statement_part

Ignored:

- file_declaration
- alias_declaration

Not supported:

- subprogram_kind
- group_template_declaration
- group_declaration

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

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

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

A subprogram statement part shall not include a wait statement.

8.2.3 Subprogram overloading

8.2.3.1 Operator overloading

Operator overloading shall be supported.

a) Signatures

Signatures shall not be supported.

Note -- In the presence of a user-defined function representing an operator (i.e. a function defined outside any of the standard packages named in Section 4), the RTL synthesis tool must produce logic matching the functionality of the user-defined function.

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
  | file_declaration
  | 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

Ignored:

- file_declaration
- alias_declaration
- disconnection_specification
- User-defined attribute declarations and their specifications, except as described in 7.1.

Not supported:

- Reserved word **package** after reserved word **end**
- shared_variable_declaration
- 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 must include the initial value expression, that is, deferred constants are not supported.

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

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

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

Ignored:

- alias_declaration
- file_declaration

Not supported:

- shared_variable_declaration
- group_template_declaration
- group_declaration
- Reserved words **package body** after reserved word **end**

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

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:

- physical_type_definition
- floating_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 7.1.1).

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 object should have an unsigned binary representation. If the range contains one or more negative values, the object should have a twos-complement implementation. The vector should have a width that is capable of representing all possible values in the range specified for the integer type definition. The synthesis tool should 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.

Note -- Integer ranges may be synthesized as if the zero value is included. For example "INTEGER **range** 9 **to** 10" may 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_type_definition
- physical_literal

Physical objects and literals other than 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 the time_expression following the reserved word **after** or the timeout_clause of a wait statement, and shall be ignored.

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 index constraint shall contain exactly one discrete range. The bounds of the discrete range shall be specified directly or indirectly as static values belonging to an integer type. The element subtype indication shall denote either a subtype of a scalar (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 name a subtype of an integer type.

In an unconstrained array definition, exactly one index subtype definition shall be supported.

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 access types 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
  | attribute_declaration
  | component_declaration
  | group_template_declaration
  | group_declaration
  | entity_declaration
  | configuration_declaration
  | subprogram_declaration
  | package_declaration
```

Supported:

- declaration

Ignored:

- alias_declaration

Not supported:

- group_template_declaration
- group_declaration

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

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
```

Supported:

- type_declaration
- full_type_declaration
- type_definition

Ignored:

- incomplete_type_declaration
- access_type_definition
- file_type_definition

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

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

Ignored:

- expression

Not supported:

- signal_kind

The initial value expression shall be ignored unless the declaration is in a package, where it 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**
- Reserved word **bus**

Generic interface constant declarations shall have a subtype indication of an integer type or a subtype thereof.

The static expression shall be ignored in port interface lists and formal parameter lists except for interface constant declarations that 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:

- function_name
- type_mark
- file_name

The formal part may be only a formal designator and the actual part shall only be an actual designator.

8.4.3.3 Alias declarations

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

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

Ignored:

- alias_declaration
- alias_designator

Not supported:

- signature

Use of aliases shall not be supported.

8.4.4 Attribute declarations

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

Ignored:

- attribute_declaration

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

8.4.5 Component declarations

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

Supported:

- component_declaration

Not supported:

- reserved word is
- component_simple_name

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

Not supported:

- signature
- Entity class group and file
- Use of user-defined attributes
- Reserved words **other** and **all** in entity_name_list

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

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
```

Ignored:

- configuration_specification
- component_specification
- instantiation_list

8.5.2.1 Binding indication

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

Ignored:

- binding_indication

Not Supported:

- generic_map_aspect
- port_map_aspect

a) Entity aspect

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

Not 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 be ignored.

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.

Only a single expression shall be permitted (no multidimensional objects).

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

Not supported:

- signature
- expression

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:

- **xnor** operator
- All shift operators

- 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

Not supported:

- **xnor** operator
- shift_operator

8.7.2.1 Logical operators

Not supported:

- **xnor** operator

8.7.2.2 Relational operators

No restriction.

Note -- Using relational operators for enumerated type that has an explicit encoding specified via the ENUM_ENCODING attribute may lead to simulation mismatches (see 7.1.1).

8.7.2.3 Shift operators

Supported:

- All SHIFT_LEFT and SHIFT_RIGHT functions defined in packages NUMERIC_BIT and NUMERIC_STD as part of IEEE Std 1076.3-1997

Not supported:

- All shift operators

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

The / (division), **mod**, and **rem** operators shall be supported only when both operands are static or when the right operand is a static power of 2.

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~~

numeric_literal ::=
    abstract_literal
  | physical_literal
```

Supported:

- literal
- numeric_literal

Not supported:

- null

References to objects and literals of type TIME may occur only within the time_expression following the reserved word **after** or the timeout_clause of a wait statement, and shall be ignored.

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

Not supported:

- record aggregates

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. 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
  | 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
- sensitivity_list
- condition_clause
- condition

Ignored:

- timeout_clause

Not Supported:

- label
- sensitivity_clause

Only one **wait until** statement shall be allowed per process statement and it shall be the first statement in the process.

Use of 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

Not supported:

- label

8.8.3 Report statement

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

```
    [severity expression] ;
```

Not Supported:

- 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

Ignored:

- delay_mechanism

Not supported:

- label
- Reserved words reject, inertial and unaffected
- 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

Not supported:

- label

8.8.5.1 Array variable assignments

Array variable assignment shall be supported.

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

Not supported:

- label

Restrictions for the actual parameter part are discribed in 8.4.3.2 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

Not supported:

- if_label

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 6.2.

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

Not supported:

- label

If a signal or variable is assigned values in some branches of a **case** statement but not in all, level-sensitive storage elements may result; see 6.2. This is true only if the assignment does not occur under the control of a clock edge.

If a metalogical 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 interpret the choice as one that may never occur. That is, the interpretation that is generated shall not be required to contain any constructs corresponding to the presence or absence of the sequence of statements associated with the choice.

Note 1 -- If the type of the case expression includes metalogical values, and if not all the metalogical values are included among the case choices, then the case statement must include an **others** choice to cover the missing metalogical choice values (IEEE Std 1076-1993).

Note 2 -- A case choice (such as "1X1") that includes a metalogical value indicates a branch that can never 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

The iteration scheme shall not be omitted.

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

Not supported:

- label

8.8.11 Exit statement

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

Supported:

- exit_statement

Not supported:

- label

8.8.12 Return statement

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

Supported:

- return_statement

Not supported:

- label

8.8.13 Null statement

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

Supported:

- null_statement

Not supported:

- label

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_clause ;] ]
  [ port_clause
  [ port_map_clause ;] ]

block_declarative_part ::=
  { block_declarative_item }

block_statement_part ::=
  { concurrent_statement }
```

Supported:

- block_statement
- block_declarative_part
- block_statement_part

Not supported:

- block_header
- guard_expression
- Reserved word **is**

8.9.2 Process statement

```
process_statement ::=
  [ process_label: ]
    [ postponed ] process [ ( sensitivity_list ) ] [ is ]
      process_declarative_part
    begin
      process_statement_part
    end 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
- alias_declaration
- User-defined attribute declarations and their specifications

Not supported:

- Reserved words **postponed** and **is**
- group_template_declaration
- group_declaration

The sensitivity list must 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 section 6.

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

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

Use of file objects, access objects (variables of access type) and aliases in a process are not 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

Ignored:

- options

Not supported:

- Reserved words postponed and guarded

Any **after** clauses shall be ignored.

Multiple waveform elements shall not be supported.

The value **unaffected** shall not be supported

Edge specifications (<clock_edge> or <clock_level>) shall not be allowed in concurrent signal assignments.

Example:

```
architecture ARCH of ENT is
begin

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

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

8.9.5.1 Conditional signal assignment

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

conditional_waveforms ::=
```

```
{ waveform when condition else }
  waveform [ when condition ]
```

Supported:

- conditional_signal_assignment
- conditional_waveforms

Ignored:

- options

Not supported:

- Last **when** condition

Conditional signal assignments that satisfy either of the following conditions shall not be supported:

a) The conditional waveforms contain a reference to one or more elements of the target signal.

b) The conditional waveforms contain an expression that represents a clock edge as defined by 6.1.2.

Example:

```
architecture ARCH of ENT 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 ARCH;
```

8.9.5.2 Selected signal assignments

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

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

Supported:

- selected_signal_assignment
- selected_waveforms

Ignored:

- options

Selected signal assignments that satisfy either of the following conditions shall not be supported:

a) The selected waveforms contain a reference to one or more elements of the target signal.

b) The selected waveforms contain an expression that represents a clock edge as defined by 6.1.2.

Examples:

```
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

Not supported:

- **entity** and **configuration** forms of instantiated unit
- reserved word component

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

Type conversions on a formal port shall not be supported.

8.9.6.1 Instantiation of a component

Component instantiation shall be supported.

8.9.6.2 Instantiation of a design entity

Not supported:

- Instantiation of a design entity

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

Not supported:

- block_declarative_item (the declarative region)
- reserved word **begin**

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} ;
```

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

No constraints shall be put on elaboration for synthesis.

8.13 Lexical elements

Real literals are only allowed in **after** clauses.

Extended identifiers shall not be supported.

8.14 Predefined language environment

8.14.1 Predefined attributes

8.14.1.1 Attributes whose prefix 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 section 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 arguments of type CHARACTER
- Functions with 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 both operands are static, or 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, or when neither operand is static
- Functions with arguments of type “universal real” or of type REAL

- Functions with arguments of type TIME
- The function NOW
- Functions with arguments of type FILE_OPEN_KIND
- Functions with arguments of type FILE_OPEN_STATUS

8.14.3 Package TEXTIO

The subprograms defined in package TEXTIO shall not be supported.

(This page left blank intentionally)

Annex A: Syntax Summary (Informative)

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

base_unit_declaration ::= identifier ;

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_clause ;] ]
  [ port_clause
  [ port_map_clause ;] ]

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

delay_mechanism ::=
    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 { , enumeraton_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_declarataion ::=
  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 ] ;

incomplete_type_declaration ::= 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
  | file_declaration

operator_symbol ::= string_literal

options ::= [ ~~guarded~~ ] [delay_mechanism]

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~~
  | file_declaration
  | 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~~
  | file_declaration
  | 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
      primary_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 ::=
    name
  | literal
  | aggregate
  | function_call
  | qualified_expression
  | type_conversion
  | ~~allocator~~
  | ( expression )

primary_unit ::=
    entity_declaration
  | configuration_declaration
  | package_declaration

primary_unit_declaration ::= identifier ;

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 process [process_label] ;

process_statement_part ::=
  { sequential_statement }

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

range ::=
    range_attribute_name
  | simple_expression direction simple_expression

range_constraint ::= range range

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

relation ::=
  shift_expression [ relational_operator shift_expression ]

relational_operator ::= = | /= | < | <= | > | >=

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

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

scalar_type_definition ::=
    enumeration_type_definition
  | integer_type_definition
  | physical_type_definition
  | floating_type_definition

secondary_unit ::=
    architecture_body
  | package_body

secondary_unit_declaration ::= identifier = physical_literal ;

selected_name ::= prefix.suffix

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

selected_waveforms ::=
  { waveform when choices , }
    waveform when choices

sensitivity_clause ::= on sensitivity_list

sensitivity_list ::= signal_name {, signal_name}

sequence_of_statements ::=
  { sequential_statement }
```

```
sequential_statement ::=
    wait_statement
  | assertion_statement
  | report_statement
  | signal_assignment_statement
  | variable_assignment
  | procedure_call_statement
  | if_statement
  | case_statement
  | loop_statement
  | next_statement
  | exit_statement
  | return_statement
  | null_statement

shift_expression ::=
  simple_expression [ shift_operator simple_expression ]

shift_operator ::= sll | srl | sla | sra | rol | ror

sign ::= + | -

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

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

signal_kind ::= register | bus

signal_list ::=
    signal_name {, signal_name }
  | others
  | all

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

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

simple_name ::= identifier

slice_name ::= prefix ( discrete_range )

string_literal ::= " { graphic_character } "

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

subprogram_declaration ::=
  subprogram_specification ;

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_declarative_part ::=
  { subprogram_declarative_item }

subprogram_kind ::= procedure | function

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

```
      return type_mark

subprogram_statement_part ::=
  { sequential_statement }

subtype_declaration ::=
  subtype identifier is subtype_indication ;

subtype_indication ::=
  [ resolution_function_name ] type_mark [ constraint ]

suffix ::=
    simple_name
  | character_literal
  | operator_symbol
  | all

target ::=
    name
  | aggregate

term ::=
  factor { multiplying_operator factor }

timeout_clause ::= for time_expression

type_conversion ::= type_mark( expression )

type_declaration ::=
    full_type_declaration
  | incomplete_type_declaration

type_definition ::=
    scalar_type_definition
  | composite_type_definition
  | access_type_definition
  | file_type_definition

type_mark ::=
    type_name
  | subtype_name

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

use_clause ::=
  use selected_name {, selected_name} ;

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

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

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

waveform ::=
    waveform_element {, waveform_element}
  | unaffected

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