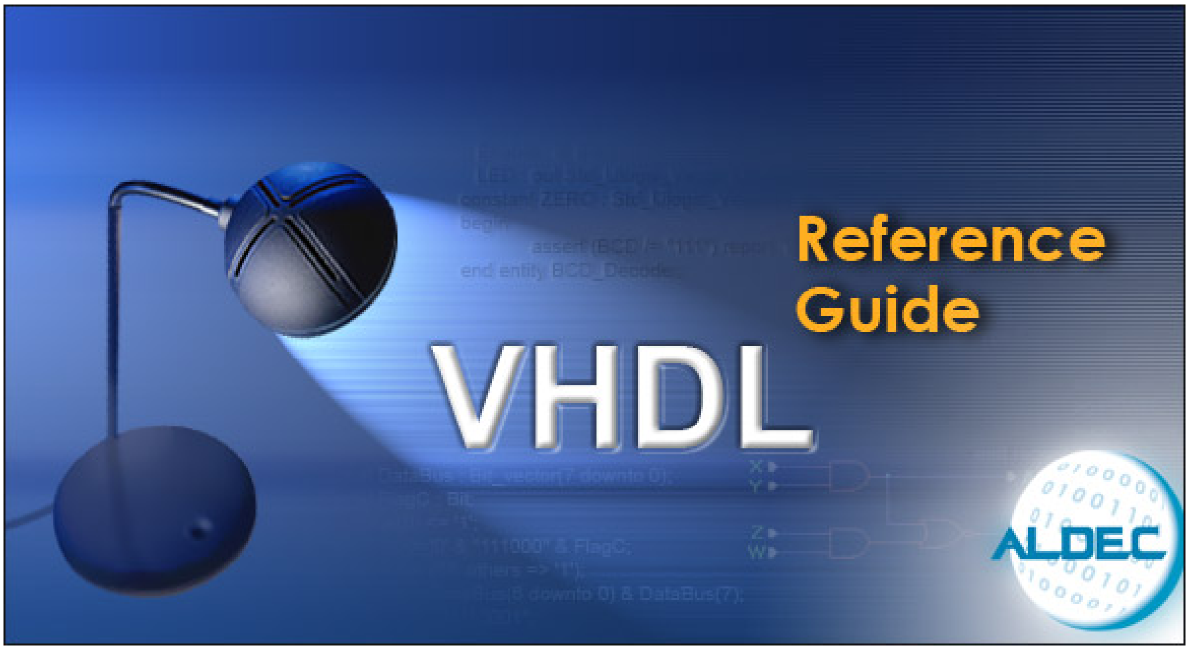

Reference Guide
VHDL
ALDEC

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Overview

The VHDL Reference Guide is divided into three sections:

- VHDL 1993
- VHDL 2002
- VHDL 2008

The VHDL 1993 section contains a complete language reference based on the IEEE 1076-1993. Links to the VHDL 2002 and VHDL 2008 sections have been added where appropriate.

The VHDL 2002 section contains only language features added to VHDL in the IEEE 1076-2002. The most significant changes compared to VHDL 1993 is the addition of protected types and adjustments related to shared variables.

The VHDL 2008 section describes features added in the latest edition of the standard, i.e. the IEEE Std 1076-2008. Constructs not yet supported by the simulator have been omitted.

The term VHDL 2008 supersedes the term VHDL 2006 used previously to informally designate the language based on subsequent drafts of the IEEE Std 1076-2008, e.g. Draft IEEE P1076-2006/D3.2.

Access Type

Formal Definition

A type that provides access to an object of a given type. Access to such an object is achieved by an access value returned by an allocator; the access value is said to designate the object.

Complete description: Language Reference Manual IEEE 1076-1993, § 3.3.

Simplified Syntax

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

Description

Access type allows to manipulate data, which are created dynamically during simulation and which exact size is not known in advance. Any reference to them is performed via allocators, which work in a similar way as pointers in programming languages.

The subtype_indication in the access type declaration denotes the type of an object designated by a value of an access type. It can be any scalar, composite or other access type (Example 1). File type is not allowed here.

The only objects allowed to be of the access type are variables.

The default value of an access type is null that designates no object at all. To assign any other value to an object of an access type an allocator has to be used (see Allocator for details).

The access type allows to create recursive data structures (dynamic lists of objects created during simulation) which consist of the records that contain elements of access types - either the same or different than the actually declared. In order to handle declarations of such recursive data structures so called incomplete type declaration is needed which plays a role of an "announcement" of a type which will be declared later on.

For each incomplete type declaration there must be a corresponding full type declaration with the same name. The complete declaration must appear in the same declarative part. A type declared as incomplete may not be used for any other purposes than to define an access type before the complete type definition is accomplished. Such an incomplete type declaration is presented in Example 2 below.

Examples

Example 1

```
-- declaration of record type Person:
type Person is record
  address : ADDRESS_TYPE;
  age : DATE;
end record;

-- declaration of access type Person_Access:
type Person_Access is access Person;
```

The `Person_Access` type defines a pointer (dynamic link) to a record declared earlier and called `Person`.

Example 2

```
-- declaration of an incomplete type Queue_Element:
type Queue_Element;

-- declaration of an access type Queue_Element_Ptr:
type Queue_Element_Ptr is access Queue_Element;

-- declaration of an full record type Queue_Element:
type Queue_Element is record
  name : STRING( 1 to 20 );
  address : ADDRESS_TYPE;
  age : DATE;
  succ : Queue_Element_Ptr;
end record;
```

The type `Queue_Element` contains the `Queue_Element_Ptr` field, which in turn points to `Queue_Element`. In order to declare both types sequentially, first the `Queue_Element` is declared in an incomplete way, which allows declaring the access type `Queue_Element_Ptr`. Finally, complete declaration of `Queue_Element` must be provided.

Important Notes

- Application of access types is restricted to variables: only variables can be of an access value. Also, only variables can be designated by an access value.
- Although access types are very useful for modeling potentially large structures, like memories or FIFOs, they are not supported by synthesis tools.

Aggregate

Formal Definition

A basic operation that combines one or more values into a composite value of a record or array type.

Complete description: Language Reference Manual IEEE 1076-1993 § 7.3.2.

Syntax

```
aggregate ::= ( element_association {, element association } )
element_association ::= [choices => ] expression
choices ::= choice { | choice}
choice ::= simple expression | discrete_range | element_simple_name | others
```

Description

An aggregate assigns one or more values to the elements of a record or array creating the composite value of this type. Aggregates are composed of element associations, which associate expressions to elements (one expression per one or more elements). Element associations are specified in parentheses and are separated by commas.

An expression assigned to an element or elements must be of the same type as the element(s).

Elements can be referred to either by textual order they have in the object declaration (so called positional associations - Example 1) or by its name (named associations - Example 2). Both methods can be used in the same aggregate, but in such a case all positional associations appear first (in textual order) and all named associations appearing next (in any order). In any case if the association `others` is used, it must be the last one in an aggregate.

The choice clause, denoting selection of element(s) can have any of the following forms:

- simple expression,
- discrete range,
- simple name, or
- reserved word `others`.

A value of simple expression can be applied in arrays only and must belong to discrete range of an array type. A simple expression specifies the element at the corresponding index value. Example 2 illustrates this concept.

A discrete range must meet the same conditions as a simple expression. It is useful when several consecutive elements of an array are assigned the same value (Example 3). A discrete range serves for defining the set of indexes only and the direction specified or implied has no significance.

The use of element simple name as a choice is restricted to records only. In this case, each element is identified by its name (Example 4).

When some elements are assigned different values and the remaining elements will receive some other value, reserved word `others` can be used to denote those elements (Example 5). Such a choice must be the last in an aggregate and can be used both in arrays and in records, provided that the remaining elements of the records are of the same type.

The choice `others` can serve as a very convenient way to assign the same value to all elements of some array, e.g. to reset a wide bus (Example 6).

If several elements are assigned the same value, a multiple choice can be used. In such a case a bar sign (|) separates references to elements (Example 7). If a multiple choice is used in an array aggregate, it may not be mixed with positional associations.

Examples

Example 1

```
variable Data_1 : BIT_VECTOR( 0 to 3 ) := ( '0', '1', '0', '1' );
```

Bits number 0 and 2 are assigned the value '0', while bits 1 and 3 are assigned '1'. All element associations here are positional.

Example 2

```
variable Data_2 : BIT_VECTOR( 0 to 3 ) := ( 1 => '1', 0 => '0', 3 => '1', 2 => '0' );
```

Like in the previous example, bits number 0 and 2 are assigned the value '0', while bits 1 and 3 are assigned '1'. The element associations here, however, are named. Note that in this case the elements can be listed in arbitrary order.

Example 3

```
signal Data_Bus : STD_LOGIC_VECTOR( 15 downto 0 );
-- ...
Data_Bus <= ( 15 => '1', 14 downto 8 => '0', 7 downto 0 => '1' );
```

Data_Bus will be assigned the value of "1000000011111111". The first element (thus it is bit number 15) is assigned a value using named association. The other two groups are assigned values using discrete ranges.

Example 4

```
type Status_Record is record
  Code : INTEGER;
  Name : STRING( 1 to 4 );
end record;

variable Status_Var : Status_Record := ( Code => 57, Name => "MOVE" );
```

Choice as an element simple name can be used in record aggregates - each element is associated a value (of the same type as the element itself).

Example 5

```
signal Data_Bus : STD_LOGIC_VECTOR( 15 downto 0 );
-- ...
Data_Bus <= ( 14 downto 8 => '0', others => '1' );
```

Data_Bus will be assigned the same value as in Example 3 i.e. "1000000011111111", but this aggregate is written in more compact way. Apart from bits 14 through 8, which receive value '0' all the others (15 and 7 through 0) will be assigned '1'. Note that the choice `others` is the last in the aggregate.

Example 6

```
signal Data_Bus : STD_LOGIC_VECTOR( 15 downto 0 );
-- ...
Data_Bus <= ( others => 'Z' );
```

Instead of assigning "ZZZZZZZZZZZZZZZZ" to Data_Bus in order to put it in high impedance state, an aggregate with the `others` choice representing all the elements can be used.

Example 7

```
signal Data_Bus : STD_LOGIC_VECTOR( 15 downto 0 );
-- ...
Data_Bus <= ( 15 | 7 downto 0 => '1', others => '0' );
```

Note the multiple choice specification of the assignment to the bits 15 and 7 through 0. The result of the assignment to Data_Bus will be the same as in Example 3 and Example 5 ("1000000011111111").

Important Notes

- Associations with elements' simple names are allowed in record aggregates only.
- Associations with simple expressions or discrete ranges as choices are allowed only in array aggregates.
- Each element of the value defined by an aggregate must be represented once and only once in the aggregate.
- Aggregates containing the single element association must always be specified using named association in order to distinguish them from parenthesized expressions.
- The others choice can be only the last in an aggregate.

Alias

Formal Definition

An alternate name for an existing named entity.

Complete description: Language Reference Manual IEEE 1076-1993 § 4.3.3.

Simplified Syntax

```
alias alias_name : alias_type is object_name;
```

Description

The alias declares an alternative name for any existing object: signal, variable, constant or file. It can also be used for "non-objects": virtually everything, which was previously declared, except for labels, loop parameters, and generate parameters.

Alias does not define a new object. It is just a specific name assigned to some existing object.

Aliases are prevalently used to assign specific names to slices of vectors in order to improve readability of the specification (see Example 1). When an alias denotes a slice of an object and no subtype indication is given then the subtype of the object is viewed as if it was of the subtype specified by the slice.

If the alias refers to some other object than a slice and no subtype indication is supported then the object is viewed in the same way as it was declared.

When a subtype indication is supported then the object is viewed as if it were of the subtype specified. In case of arrays, the subtype indication can be of opposite direction than the original object (Example 2).

Subtype indication is allowed only for object alias declarations.

A reference to an alias is implicitly a reference to the object denoted by the alias (Example 3).

Examples

Example 1

```
signal Instruction : BIT_VECTOR( 15 downto 0 );
alias Op_Code : BIT_VECTOR( 3 downto 0 ) is Instruction( 15 downto 12 );
alias Source : BIT_VECTOR( 1 downto 0 ) is Instruction( 11 downto 10 );
alias Destination : BIT_VECTOR( 1 downto 0 ) is Instruction( 9 downto 8 );
alias Immediate_Data : BIT_VECTOR( 7 downto 0 ) is Instruction( 7 downto 0 );
```

The four aliases in the example above denote four elements of an instruction: operation code, source code, destination code and immediate data supported for some operations. Note that in all declarations the number of bits in the subtype indication and the subtype of the original object match.

Example 2

```
signal Data_Bus : BIT_VECTOR( 31 downto 0 );
alias First_Nibble : BIT_VECTOR( 0 to 3 ) is Data_Bus( 31 downto 28 );
```

`Data_Bus` and `First_Nibble` have opposite directions. A reference to `First_Nibble( 0 to 1 )` is equivalent to a reference to `Data_Bus( 31 downto 30 )`.

Example 3

```
signal Instruction : BIT_VECTOR( 15 downto 0 );
alias Op_Code : BIT_VECTOR( 3 downto 0 ) is Instruction( 15 downto 12 );
-- ...
if Op_Code = "0101" -- equivalent to if Instruction( 15 downto 12 ) = "0101"
then
-- ...
```

Both conditions are exactly the same, but the one where alias is used is more legible.

Important Notes

- VHDL Language Reference Manual uses the name 'entity' to denote a language unit, i.e. object, parameter etc. It is completely different idea than a design entity.
- Many synthesis tools do not support aliases.

Allocator

Formal Definition

An operation used to create anonymous, variable objects accessible by means of access values.

Complete description: Language Reference Manual IEEE 1076-1993 § 7.3.6.

Simplified Syntax

```
new subtype_indication
new qualified_expression
```

Description

Each time an allocator is evaluated, a new object is created and the object is designated (pointed) by an access value (pointer). The type of the object created by an allocator is defined either by a subtype indication (Example 1 and Example 2) or a qualified expression (Example 3 and Example 4).

In case of allocators with a subtype indication, the initial value of the created object is the same as the default initial value of a directly declared variable of the same subtype (Example 1 and Example 2). When qualified expression is used, the initial value is defined by the expression itself (Example 3 and Example 4).

If an allocator creates an object of an array type, then the array must be constrained. This can be achieved using a constrained subtype or specifying an explicit index constraint in the subtype indication (Example 2).

See also Access Type.

Examples

Example 1

```
type Table is array ( 1 to 8 ) of NATURAL;
type Table_Access is access Table;
variable y : Table_Access;
-- ...
y := new Table; -- will be initialized with
                -- ( 0, 0, 0, 0, 0, 0, 0, 0 )
```

The allocator (note that the allocator is of the access type) creates a new object of the Table type, which is initialized to a default value, equal in this case to (0, 0, 0, 0, 0, 0, 0, 0).

Example 2

```
z := new BIT_VECTOR( 1 to 3 );
```

This allocator creates a new object of the `BIT_VECTOR` type, consisting of three elements. The default initial value of this object is equal to ('0', 0', 0'). Note that the subtype indication is constrained as the `BIT_VECTOR` type is unconstrained.

Example 3

```
type test_record is record
  test_time : TIME;
  test_value : BIT_VECTOR( 0 to 3 );
end record;

type Access_Test_Record is access test_record;
variable x, z : Access_Test_Record;

x := new test_record'( 30 ns, B"1100" ); -- record allocation with aggregate
z := new test_record;
z.test_time := 30 ns;
z.test_value := B"1100";
```

Initial values can be assigned to an object (in this case a record) created by an allocator both using a qualified expression (in this case with an aggregate - allocator for x) or using a subtype indication and later on direct assignments (allocator for z). In both cases above the objects created will be identical (although it will not be the same object).

Example 4

```
type access_BIT_VECTOR is access BIT_VECTOR( 7 downto 0 );
variable Ptr1, Ptr2 : access_BIT_VECTOR;
Ptr1 := new BIT_VECTOR( 7 downto 0 );
Ptr2 := Ptr1;
```

There is no allocator assigned to `Ptr2`, thus no new object will be created for it. Instead it will point to the same object, which was created for `Ptr1`.

Important Notes

- For each access type an implicitly declared procedure `deallocate` is defined. The procedure reverses the evaluation of an allocator, i.e. releases the storage occupied by an object created by an allocator.
- Allocators (and access types) are not synthesizable.
- A subtype indication in allocator must not include a resolution function.
- An object created by an allocator has not its own name (indicator). Instead, it is referred to through the name, which it was allocated to.
- The concept of access types and allocators is very much the same as the concept of pointers in software programming languages.

Architecture

Formal Definition

A body associated with an entity declaration to describe the internal organization or operation of a design entity. An architecture body is used to describe the behavior, data flow, or structure of a design entity.

Complete description: Language Reference Manual IEEE 1076-1993 § 1.2.

Simplified Syntax

```
architecture architecture_name of entity_name is
  architecture_declarations
begin
  concurrent_statements
end [ architecture ] [ architecture_name ];
```

Description

Architecture assigned to an entity describes internal relationship between input and output ports of the entity. It consists of two parts: declarations and concurrent statements.

First (declarative) part of an architecture may contain declarations of types, signals, constants, subprograms (functions and procedures), components, and groups. See respective topics for details.

Concurrent statements in the architecture body define the relationship between inputs and outputs. This relationship can be specified using different types of statements: concurrent signal assignment, process statement, component instantiation, concurrent procedure call, generate statement, concurrent assertion statement and block statement. It can be written in different styles: structural, dataflow, behavioral (functional) or mixed.

The description of a structural body is based on component instantiation and generate statements. It allows to create hierarchical projects, from simple gates to very complex components, describing entire subsystems. The connections among components are realized through ports. Example 1 illustrates this concept for a BCD decoder.

The Dataflow description is built with concurrent signal assignment statements. Each of the statements can be activated when any of its input signals changes its value. While these statements describe the behavior of the circuit, a lot of information about its structure can be extracted form the description as well. Example 2 contains this type of description for the same BCD decoder as in the previous example.

The architecture body describes only the expected functionality (behavior) of the circuit, without any direct indication as to the hardware implementation. Such description consists only of one or more processes, each of which contains sequential statements (Example 3).

The architecture body may contain statements that define both behavior and structure of the circuit at the same time. Such architecture description is called mixed (Example 4).

Examples

Example 1

```
architecture Structure of Decoder_bcd is
  signal S: BIT_VECTOR( 0 to 1 );

  component AND_Gate
    port(
      A, B : in BIT;
      D : out BIT
    );
  end component;

  component Inverter
    port(
      A : in BIT;
      B : out BIT
    );
  end component;
begin
  Inv1: Inverter port map( A => bcd(0), B => S(0) );
  Inv2: Inverter port map( A => bcd(1), B => S(1) );
  A1: AND_Gate port map( A => bcd(0), B => bcd(1), D => led(3) );
  A2: AND_Gate port map( A => bcd(0), B => S(1), D => led(2) );
  A3: AND_Gate port map( A => S(0), B => bcd(1), D => led(1) );
  A4: AND_Gate port map( A => S(0), B => S(1), D => led(0) );
end architecture;
```

The components `Inverter` and `AND_Gate` are instantiated under the names `Inv1`, `Inv2`, `A1`, `A2`, `A3`, and `A4`. The connections among the components are realized by the use of signals `S(0)`, `S(1)` declared in the architecture's declarative part.

Example 2

```
architecture Dataflow of Decoder_bcd is
begin
  led(3) <= bcd(0) and bcd(1);
  led(2) <= bcd(0) and ( not bcd(1) );
  led(1) <= ( not bcd(0) ) and bcd(1);
  led(0) <= ( not bcd(0) ) and ( not bcd(1) );
end architecture;
```

All the four statements here are executed concurrently and each of them is activated individually when any of its input signals changes its value.

Example 3

```
architecture procedural of Decoder_bcd is
  signal S: BIT_VECTOR( 3 downto 0 );
begin
  P1: process ( bcd, S )
  begin
    case bcd is
      when "00" => S <= "0001";
      when "01" => S <= "0010";
      when "10" => S <= "0100";
      when "11" => S <= "1000";
    end case;
    led <= S;
  end process;
end architecture;
```

In the `P1` process, both intermediate signal `S` and output signal `led` are assigned. The `led <= S;` signal assignment could be moved out of the `P1` process scope and signal `S` removed from the process' Sensitivity List.

Example 4

```
architecture Mixed of Decoder_bcd is
  signal S: BIT_VECTOR( 0 to 2 );

  component Inverter
    port(
      A: in BIT;
      B: out BIT
    );
  end component;
begin
  Inv1: Inverter port map( A => bcd(0), B => S(0) );
  Inv2: Inverter port map( A => bcd(1), B => S(1) );
  P: process ( S, bcd )
  begin
    led(0) <= S(0) and S(1);
    led(1) <= S(0) and bcd(1);
    led(2) <= bcd(0) and S(1);
    led(3) <= bcd(0) and bcd(1);
  end process;
end architecture;
```

Above, two `Inverter` component instantiation statements define the circuit responsible for determining the value of the signal `S`. This signal is read by behavioral part i.e. the process statement `P`. In this process, the values computed by the and operation are assigned to the `led` output port.

Important Notes

- Single entity can have several architectures, but an architecture cannot be assigned to different entities.
- An architecture may not be used without an entity.
- All declarations defined in an entity are fully visible and accessible within each architecture assigned to this entity.
- Different types of statements (i.e. processes, blocks, concurrent signal assignments, component instantiations, etc.) can be used in the same architecture.

Array

Formal Definition

A type, the value of which consists of elements that are all of the same subtype (and hence, of the same type). Each element is uniquely distinguished by an index (for a one-dimensional array) or by a sequence of indexes (for a multidimensional array). Each index must be a value of a discrete type and must lie in the correct index range.

Complete description: Language Reference Manual IEEE 1076-1993 § 3.2.1.

Simplified Syntax

```
type type_name is array ( range_spec ) of element_type
type type_name is array ( range_type_name range <> ) of element_type
```

Description

An array is a composite object, which elements are of the same subtype. Each of the elements is indexed by one or more indices belonging to specified discrete types. The number of indices is the number of dimensions, i.e. one-dimensional array has one index, two-dimensional has two indices, etc. The order of indices is significant and follows the order of dimensions in the type declaration (Example 1).

An array may be either constrained or unconstrained. The array is constrained if the size of the array is constrained. The size of the array can be constrained using a discrete type mark or a range. In both cases, the number of the elements in the array is known during the compilation. Several declarations of constrained arrays are presented in Example 2.

The array is said to be unconstrained if its size is unconstrained: the size of the unconstrained array is declared in the form of the name of the discrete type, which range is unconstrained. The number of elements of unconstrained array type is unknown. The size of a particular object is specified only when it is declared. Example 2 presents several declarations of unconstrained arrays.

Package STANDARD contains declarations of two one-dimensional unconstrained predefined array types: `STRING` and `BIT_VECTOR`. The elements of the `STRING` type are of the type `CHARACTER` and are indexed by positive values (i.e. counted from 1), and the elements of the `BIT_VECTOR` type are of the type `BIT` and are indexed by natural values (i.e. counted from 0). See STRING type and BIT_VECTOR for details.

Array elements are referenced by indices and can be assigned values individually or using concatenation, aggregates, slices or any mixture of those methods. See respective topics for details.

Examples

Example 1

```
type Real_Matrix is array ( 1 to 10 ) of REAL;
type BYTE is array ( 0 to 7 ) of BIT;
type Log_4_Vector is array ( POSITIVE range 1 to 8, POSITIVE range 1 to 2 ) of Log_4;
type X is ( LOW, HIGH );
type DATA_BUS is array ( 0 to 7, X ) of BIT;
```

The type `Real_Matrix` is an array consisting of 10 elements, each of which is of the type `REAL`. `Log_4_Vector` is a two-dimensional array 8x2 and its elements are of type `Log_4` (which must have been declared earlier). Also the type `DATA_BUS` is a two-dimensional array of the same size, but note that one of the dimensions is defined as enumeration type `X`.

Example 2

```
-- unconstrained array of element of REAL type:
type Real_Matrix is array ( POSITIVE range <> ) of REAL;
variable Real_Matrix_Object : Real_Matrix ( 1 to 8 );

-- unconstrained array of elements of Log_4 type:
type Log_4_Vector is array ( NATURAL range <>, POSITIVE range<> ) of Log_4;
variable L4_Object : Log_4_VECTOR( 0 to 7, 1 to 2 );
```

Examples of unconstrained types: `Real_Matrix` is when an unconstrained type and an object of this type is declared (`Real_Matrix_Object`) it is restricted to 8 elements. In similar way `L4_Object` is constrained from an unconstrained two-dimensional type `Log_4_Vector`.

Important Notes

- Synthesis tools do generally not support multidimensional arrays. The only exceptions to this are two-dimensional "vectors of vectors". Some synthesis tools allow two-dimensional arrays.
- Arrays may not be composed of files.

Assertion Statement

Formal Definition

A statement that checks that a specified condition is true and reports an error if it is not.

Complete description: Language Reference Manual IEEE 1076-1993 § 8.2, § 9.4.

Simplified Syntax

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

Description

The assertion statement has two of three parts optional. Usually all three are used.

The condition specified in an assertion statement must evaluate to a BOOLEAN value (TRUE or FALSE). If it is FALSE, it is said that an assertion violation occurred.

The expression specified in the `report` clause must be of predefined type STRING and is a message to be reported when assertion violation occurred. See also Report Statement.

If the `severity` clause is present, it must specify an expression of predefined type SEVERITY_LEVEL, which determines the severity level of the assertion violation. The SEVERITY_LEVEL type is specified in the STANDARD package and contains following values: NOTE, WARNING, ERROR, and FAILURE. If the `severity` clause is omitted it is implicitly assumed to be ERROR.

When an assertion violation occurs, the report is issued and displayed on the screen. The supported severity level supplies an information to the simulator. The severity level defines the degree to which the violation of the assertion affects operation of the process:

- NOTE can be used to pass information messages from simulation (Example 1);
- WARNING can be used in unusual situation in which the simulation can be continued, but the results may be unpredictable (Example 2);
- ERROR can be used when assertion violation makes continuation of the simulation not feasible (Example 3);

Assertion statements are not only sequential, but can be used as concurrent statements as well. A concurrent assertion statement represents a passive process statement containing the specified assertion statement.

Examples

Example 1

```
assert Status = OPEN_OK
  report "The call to FILE_OPEN was not successful"
  severity WARNING;
```

Having called the procedure FILE_OPEN, if the status is different from OPEN_OK, it is indicated by the warning message.

Example 2

```
assert not ( ( S = '1' ) and ( R = '1' ) )
  report "Both values of signals S and R are equal to '1'"
    severity ERROR;
```

When the values of the signals S and R are equal to '1', the message is displayed and the simulation is stopped because the severity is set to ERROR.

Example 3

```
assert Operation_Code = "0000"
  report "Illegal Code of Operation"
    severity FAILURE;
```

Event like illegal operation code are severe errors and should cause immediate termination of the simulation, which is forced by the severity level FAILURE.

Important Notes

- The message is displayed when the condition is NOT met, therefore the message should be an opposite to the condition.
- Concurrent assertion statement is a passive process and as such can be specified in an entity.
- Concurrent assertion statement monitors specified condition continuously.
- Synthesis tools generally ignore assertion statements.

Attributes (predefined)

Formal Definition

A value, function, type, range, signal, or constant that may be associated with one or more named entities in a description.

Complete description: Language Reference Manual IEEE 1076-1993 § 14.1.

Simplified Syntax

```
object'attribute_name
```

Description

Attributes allow retrieving information about named entities: types, objects, subprograms etc. VHDL standard defines a set of predefined attributes. Additionally, users can define new attributes, and then assign them to named entities by specifying the entity and the attribute values for it. See attributes (user-defined) for details.

Predefined attributes denote values, functions, types, and ranges that characterize various VHDL entities. Separate sets of attributes are predefined for types, array objects or their aliases, signals and named entities.

Each type or subtype T has a basic attribute called T'Base, which indicates the base type for type T (Table 1). It should be noted that this attribute could be used only as a prefix for other attributes.

Table 1. Attributes available for all types

Attribute	Result
T'Base	base type of T

Scalar types have attributes, which are described in the Table 2. Letter T indicates the scalar type.

Table 2. Scalar type attributes

Attribute	Result type	Result
T'Left	same as T	leftmost value of T
T'Right	same as T	rightmost value of T
T'Low	same as T	least value in T
T'High	same as T	greatest value in T
T'Ascending	`BOOLEAN`	`TRUE` if T is an ascending range, `FALSE` otherwise
T'Image(x)	`STRING`	a textual representation of the value x of type T

T'Value(s)	base type of T	value in T represented by the string s

Discrete or physical types and subtypes additionally have attributes, which are described in Table 3. The discrete or physical types are marked with letter T before their names.

Table 3. Attributes of discrete or physical types and subtypes

Attribute	**Result type**	**Result**
T'Pos(s)	universal integer	position number of s in T
T'Val(x)	base type of T	value at position x in T (x is integer)
T'Succ(s)	base type of T	value at position one greater than s in T
T'Pred(s)	base type of T	value at position one less than s in T
T'Leftof(s)	base type of T	value at position one to the left of s in T
T'Rightof(s)	base type of T	value at position one to the right of s in T

Array types or objects of the array types have attributes, which are listed in the Table 4. Aliases of the array type objects have the same attributes. Letter A denotes the array type or array objects below.

Table 4. Attributes of the array type or objects of the array type

Attribute	**Result**
A'Left(n)	leftmost value in index range of dimension n
A'Right(n)	rightmost value in index range of dimension n
A'Low(n)	lower bound of index range of dimension n
A'High(n)	upper bound of index range of dimension n
A'Range(n)	index range of dimension n
A'Reverse_range	reversed index range of dimension n
A'Length(n)	number of values in the n-th index range
A'Ascending(n)	TRUE if index range of dimension n is ascending, FALSE otherwise

Signal attributes are listed in Table 5. Letter S indicates the signal names.

Table 5. Signals attributes

Attribute	**Result**
S'Delayed(t)	implicit signal, equivalent to signal S, but delayed t units of time
S'Stable(t)	implicit signal that has the value True when no event has occurred on S for t time units, False otherwise

S'Quiet(t)	implicit signal that has the value True when no transaction has occurred on S for t time units, False otherwise
S'Transaction	implicit signal of type Bit whose value is changed in each simulation cycle in which a transaction occurs on S (signal S becomes active)
S'Event	TRUE if an event has occurred on S in the current simulation cycle, False otherwise
S'Active	TRUE if a transaction has occurred on S in the current simulation cycle, FALSE otherwise
S'Last_event	the amount of time since last event occurred on S, if no event has yet occurred it returns Time'High
S'Last_active	the amount of time since last transaction occurred on S, if no event has yet occurred it returns Time'High
S'Last_value	the previous value of S before last event occurred on it
S'Driving	TRUE if the process is driving S or every element of a composite S, or FALSE if the current value of the driver for S or any element of S in the process is determined by the null transaction
S'Driving_value	the current value of the driver for S in the process containing the assignment statement to S

The named entities have attributes described in Table 6. Letter E denotes the named entities.

Table 6. Attributes of named entities

Attribute	**Result**
E'Simple_name	a string representing the simple name, character literal or operator symbol defined in the declaration of the item E
E'Path_name	a string describing the path through the design hierarchy, from the root entity or package to the item E
E'Instance_name	a string describing the path through the design hierarchy, from the root entity or package to the item E, but including the names of the entity and architecture bound to each component instance in the path

Paths which can be written using E'Path_name and E'Instance_name are used for reporting and assertion statements. They allow specifying precisely where warnings or errors are generated. E'Simple_name attribute refers to all named entities, E'Path_name and E'Instance_name can refer to all named entities apart from the local ports and generic parameters in the component declaration.

There is one more predefined attribute: 'Foreign' that allows the user to transfer additional information to the simulator. The information contains the instruction for special treatment of a given named entity. The exact interpretation of this attribute, however, depends on its implementation in particular simulator.

Examples

Example 1

```
type Table is array ( 1 to 8 ) of BIT;
variable Array_1 : Table := "10001111";
```

Array_1'Left, the leftmost value in index range of Table array, is equal to 1.

Example 2

```
type Table is array ( POSITIVE range <> ) of BIT;
subtype Table_New is Table ( 1 to 4 );
```

Table_New'Base, the base type of the Table_New subtype is Table.

Example 3

```
type New_Range is range 1 to 10;
```

New_Range'Ascending is TRUE (the New_Range type is of ascending range).

Example 4

```
type New_Values is ( Low, High, Middle );
```

New_Values'Pred(High) will bring the 'Low' value.

Example 5

```
type Table is array ( 1 to 8 ) of BIT;
```

Table'Range(1) is the range of the first index of Table type and returns '1 to 8'. Table'Range will have the same interpretation for one dimensional array.

Important Notes

- Not all predefined attributes are supported by synthesis tools; most tools support 'high, 'low, 'left, 'right, range, 'reverse_range, 'length and 'event. Some also support 'last_value and 'stable.
- See Attributes (predefined) in the VHDL 2008 section for other attributes supported in IEEE Std 1076-2008.

Attributes (user-defined)

Formal Definition

A value, function, type, range, signal, or constant that may be associated with one or more named entities in a description.

Complete description: Language Reference Manual IEEE 1076-1993 § 4.4, § 5.1.

Simplified Syntax

```
attribute attribute_name : type; -- attribute declaration
attribute attribute_name of item : item_class is expression; -- attribute specification
```

Description

VHDL allows attaching additional information to design elements through new attributes for specified types. In order to assign an attribute to a given design element, attribute specification is used. The values assigned this way can be referred in the expressions through declared attribute name.

The attribute declaration defines a new attribute within the scope of the given declarative area. It consists of an identifier specification, which represents user defined attribute and type mark that indicates value type for this attribute. A user-defined attribute can be of any VHDL type, except for an access type, file type, and any complex type with elements of any of the two types. See Example 1.

Attribute specification assigns an attribute declared earlier to a chosen named entity. The named entities that can be assigned attributes are: entity, architecture, configuration, procedure, function, package, type, subtype, constant, signal, variable, component, label, literal, units, group, or file. The named entities are enumerated in entity names list. In such a way the attribute is assigned to a given language unit. Finally, the attribute specification contains an expression, which sets an attribute value for the entities listed in the specification. See Example 2.

The attribute specification for most named entities must be declared together with declarations of those entities. For some entities, however, the attribute specification is written in other places.

The attribute specification for library units such as entity, architecture configuration and package cannot be directly placed in the library, which contains library unit declarations. Because of that, the attribute specification is placed in the declaration part of named entity (Example 3).

The attribute specification for a subprogram> must be declared in the same visibility region as the subprogram's declaration. In case of overloaded procedures and functions, signatures must be used to point to the subprogram to which the attribute is assigned (Example 4). If no signature is used the attribute relates to all subprograms with the same name.

Functions declared as the operators are always overloaded, and that is why they always require the signature in the attribute specification to differentiate functions (Example 5).

Attribute specifications for ports and generic parameters> are placed in the declaration part of the design entity or the block statement to which they belong. The attribute specifications for formal parameters of subprograms are placed in the declaration part of these subprograms (Example 6).

Attribute specifications for labeled statements> cannot be located directly in the place of the statements' declarations. Therefore, the attribute specifications for the label connected with any concurrent or sequential statements are placed in the declaration part before the occurrence of a given statement (Example 7).

In case of the sequential statement labels, the attribute specification is placed in the declaration part of the process or subprogram.

The attribute specification for literal> must be declared in the same visibility block as the literal declaration. In case when there are several literals with the same names, the attribute specification for a given literal uses a signature in order to distinguish which type a given literal belongs to (Example 8). If no signature is used the attribute can be applied to all literals with the same name.

When specifying attributes reserved words: `others` and `all` can be used as entity (item) names. In the first case, the attribute specification refers to all the remaining visible named entities of a given entity class which do not have the attribute value assigned to them. Such an attribute specification must be the last in the declaration that refers to this attribute. When the keyword `all` is used, the attribute specification refers to all named entities of the given class. Such an attribute specification must be the first in the declaration part, which relates to this attribute. See Example 9.

Examples

Example 1

```
package Attribute_package is
  attribute Component_symbol : STRING;
  attribute Pin_code : POSITIVE;
  attribute Max_delay: TIME;

  type Point is record
    x, y: REAL;
  end record;
  attribute Coordinate : Point;
end package;
```

The `Attribute_package` contains several attribute declarations, which can be later specified and used in other design units.

Example 2

```
package Some_declarations is
use Work.Attribute_package.Component_symbol,
  Work.Attribute_package.Coordinate,
  Work.Attribute_package.Pin_code,
  Work.Attribute_package.Max_delay;
constant Const_1: POSITIVE := 10;
signal Sig_1: BIT_VECTOR( 0 to 31 );
component Comp_1 is
  port(
    -- ...
  );
end component;
attribute Component_symbol of Comp_1 : component is "Counter_16";
attribute Coordinate of Comp_1 : component is ( 0.0, 17.5 );
attribute Pin_code of Sig_1 : signal is 17;
attribute Max_delay of Const_1 : constant is 10 ns;
  -- ...
end package;
```

The package `Some_declarations` specifies attributes, which were declared in the package `Attribute_package`.

Example 3

```
package Test_pkg is
attribute Package_attribute : STRING;
attribute Package_attribute of Test_pkg : package is "Training_package";
  -- ...
end package;
```

The specification of a package attribute `Package_attribute` for the package `Test_pkg` is declared in the declarative part of the package.

Example 4

```
procedure Sub_values ( a, b : in INTEGER; result: out INTEGER );
procedure Sub_values ( a, b : in BIT_VECTOR; result: out BIT_VECTOR );
attribute Description : STRING;
attribute Description of Sub_values [INTEGER, INTEGER, INTEGER] : procedure is "integer_sub_values";
attribute Description of Sub_values [BIT_VECTOR, BIT_VECTOR, BIT_VECTOR] : procedure is
"bit_vector_sub_values";
```

The specification of the attribute `Description` intended for overloaded procedure `Sub_values`, which subtracts two values of INTEGER or BIT_VECTOR types, requires signature specification. These signatures (simplified parameter lists) distinguishes versions of the procedure.

Example 5

```
function "-" ( a, b : New_logic ) return New_logic;
attribute Characteristic : STRING;
attribute Characteristic of "-" [ New_logic, New_logic return New_logic] : function is
"New_logic_op";
```

To identify operator "-" overloaded for two values of type `New_logic` it is necessary to use signature that will unambiguously identify the overloaded function.

Example 6

```
procedure Insert ( fifo : inout Fifo_type; element: in Elem_type ) is
  attribute Number of fifo : variable is 50;
  attribute Trace of element : constant is "Integer/Decimal";
  -- ...
end procedure;
```

The procedure `Insert` has two formal parameters of different classes. Specifications of attributes `Number` and `Trace` for parameters `fifo` and `element`, respectively, are placed in the declarative part of the procedure.

Example 7

```
architecture Struct of ALU is
  component Adder is
    port(
      -- ...
    )
  end component;
  attribute Coordinate of the_Adder : label is ( 0.0, 0.12 );
begin
  the_Adder : Adder port map(
    -- ...
  );
  -- ...
end architecture;
```

Specification of the attribute `Coordinate` for the label `the_Adder` for component instantiation statement is located in the declarative part of the corresponding architecture body `Struct`.

Example 8

```
type Three_level_logic is ( Low, High, Idle );
type Four_level_logic is ( Low, High, Idle, Uninitialized );
attribute Hex_value : STRING( 1 to 2 );
attribute Hex_value of Low [return Four_level_logic] : literal is "F0";
attribute Hex_value of High [return Four_level_logic] : literal is "F1";
attribute Hex_value of Idle [return Four_level_logic] : literal is "F2";
attribute Hex_value of Uninitialized : literal is "F3";
```

As the literals `Low`, `High`, `Idle` are overloaded, it is necessary to use signature indicating their type in the specification of the attribute `Hex_value` for these literals. However, this is not necessary for the literal `Uninitialized` as it is not overloaded.

Example 9

```
B1: block
  signal S1, S2, S3: STD_LOGIC;
  attribute Delay_attribute : TIME;
  attribute Delay_attribute of all : signal is 100 ps;
begin
  -- ...
end block;
```

The `Delay_attribute` relates to all signals in the block `B1`.

Important Notes

- Common attributes can be declared for objects of different classes using one construct - Group.

BIT

Formal Definition

The BIT type is predefined in the STANDARD package as an enumerated data type with only two values: '0' and '1'.

Syntax

```
type BIT is ( '0', 1' );
```

Description

The BIT type is the basic type to represent logical values. Note that there are only two values defined for the BIT type and it is not possible to use it for high impedance and other non-trivial values such as Unknown, Resistive Weak, etc. (see STD_LOGIC).

According to the type definition, its leftmost value is '0', therefore the default value of any object of the BIT type is '0'.

As the BIT type is defined in the STANDARD package, it can be used in any VHDL specification without additional declarations.

Signals of the BIT type are not resolved which means that such a signal can be assigned an expression only once in the entire architecture.

Examples

Example 1

```
signal BitSig1, BitSig2 : BIT;
-- ...
BitSig1 <= '1';
BitSig2 <= not BitSig1;
```

The BitSig1 and BitSig2 signals are declared without an initial value, therefore by default they will be assigned the '0' value. In the next statement BitSig1 is assigned the '1' value. This value is complemented in the following statement and is assigned to BitSig2. Any additional assignment either to BitSig1 or BitSig2 would be illegal.

Important Notes

- Unlike in traditional ("hand-based") digital design, logical values 0 and 1 (BIT type values '0' and '1') are NOT identical to BOOLEAN values (FALSE and TRUE), respectively. In VHDL, the latter items form completely different type (BOOLEAN).

- Logical values for object of the `BIT` type MUST be written in single quotes to distinguish them from integer values.

BIT_VECTOR

Definition

The BIT_VECTOR type is predefined in the STANDARD package as a standard one-dimensional array type with each element being of the BIT type.

Complete description: Language Reference Manual IEEE 1076-1993, § 3.2.1.2.

Syntax

```
type BIT_VECTOR is array ( NATURAL range <> ) of BIT;
```

Description

The BIT_VECTOR type is an unconstrained vector of elements of the BIT type. The size of a particular vector is specified during its declaration (see the example below). The way the vector elements are indexed depends on the defined range and can be either ascending or descending (see Range).

Assignment to an object of the BIT_VECTOR type can be performed in the same way as in case of any arrays, i.e. using single element assignments, concatenation, aggregates, slices or any combination of them.

Examples

Example 1

```
signal Data_Bus : BIT_VECTOR( 7 downto 0 );
signal Flag_C : BIT;
Data_Bus(0) <= '1'; -- 1
Data_Bus <= '0' & "111000" & Flag_C; -- 2
Data_Bus <= ( '0', others => '1' ); -- 3
Data_Bus <= Data_Bus( 6 downto 0 ) & Data_Bus(7); -- 4
Data_Bus <= "01110001"; -- 5
```

There is one BIT_VECTOR defined in this example - `Data_Bus`. Its range is defined as descending, therefore the most significant bit will be `Data_Bus(7)`. Line 1, marked in the comment field, illustrates assignment of a single element (bit). The line 2 shows typical use of a concatenation. Note that both single bits, groups of bits (with double quotes!) and other signals (as long as their type is compatible) can be used. The line 3 demonstrates the use of aggregates. The line 4 illustrates how slices can be used together with a concatenation. The value of `Data_Bus` will be rotated left in this example. Finally, in line 5 `Data_Bus` is assigned an explicit value, specified with double quotes.

Despite that each of the numbered lines above is correct, it would be illegal to put them together in one specification as shown above, due to the fact that BIT_VECTOR is an unresolved type and there can be only one assignment to an object of this type in an architecture.

Important Notes

- Logical values for objects of the BIT_VECTOR type MUST be written in double quotes. Single elements, however, are of the BIT type, therefore all values assigned to single elements are specified in single quotes.

Block Statement

Formal Definition

The block statement is a representation of design or hierarchy section, used for partitioning architecture into self-contained parts.

Complete description: Language Reference Manual IEEE 1076-1993 § 9.1.

Simplified Syntax

```
block_label : block [( guard_condition )] [is]
  declarations
begin
  concurrent statements
end block [block_label];
```

Description

The block statement is a way of grouping concurrent statements in an architecture. There are two main purposes for using blocks: to improve readability of the specification and to disable some signals by using the guard expression (see Guard for details).

The main purpose of block statement is organizational only - introduction of a block does not directly affect the execution of a simulation model. For example, both the upper and lower sections of code in Example 1 will generate the same simulation results.

Each block must be assigned a label placed just before the `block` reserved word. The same label may be optionally repeated at the end of the block, right after the `end block` reserved words.

A block statement can be preceded by two optional parts: a header and a declarative part. The latter allows to introduce declarations of subprograms, types, subtypes, constants, signals, shared variables, files, aliases, components, attributes, configurations, disconnections, use clauses and groups (i.e. any of the declarations possible for an architecture). All declarations specified here are local to the block and are not visible outside of it.

A block header may contain port and generic declarations (like in an entity), as well as so called port map and generic map declarations. The purpose of port map and generic map statements is to map signals and other objects declared outside of the block into the ports and generic parameters that have been declared inside the block, respectively. This construct, however, has only a small practical importance. The Example 2 illustrates typical block declarations.

If an optional guard condition is specified at the beginning of the block then this block becomes a guarded block. See Guard for details.

The statements part may contain any concurrent constructs allowed in an architecture. In particular, other block statements can be used here. This way, a kind of hierarchical structure can be introduced into a single architecture body.

Examples

Example 1

```
A1: OUT1 <= '1';
LEVEL1 : block
begin
  A2: OUT2 <= '1';
  A3: OUT3 <= '0';
end block LEVEL1;
-- vs.
A1: OUT1 <= '1';
A2: OUT2 <= '1';
A3: OUT3 <= '0';
```

Both pieces of code above will behave in exactly the same way during simulation - block construct only separates part of the code without adding any functionality.

Example 2

```
entity X_GATE is
  generic(
    Long_Time : TIME;
    Short_Time : TIME
  );
  port(
    P1 : inout BIT;
    P2 : inout BIT;
    P3 : inout BIT
  );
end entity;

architecture STRUCTURE of X_GATE is
  -- global declarations of signals
  signal A, B : BIT;
begin
  LEVEL1 : block
    -- local declaration of generic parameters
    generic(
      GB1 : TIME;
      GB2 : TIME
    );
    -- local binding of generic parameters
    generic map(
      GB1 => Long_Time,
      GB2 => Short_TIME
    );
    -- local declaration of ports
    port(
      PB1 : in BIT;
      PB2 : inout BIT
    );
    -- local binding of ports and signals
    port map(
      PB1 => P1,
      PB2 => B
    );
    -- local declarations:
    constant Delay : TIME := 1 ms;
    signal S1 : BIT;
  begin
    S1 <= PB1 after Delay;
    PB2 <= S1 after GB1, P1 after GB2;
  end block;
end architecture;
```

The signals PB1 and PB2 have here the same values as P1 and B (in port map statement), respectively, and the generics GB1 and GB2 (see generic map statement) have the same values as Long_Time and Short_Time, respectively. However, such assignment is redundant because a block may use any declarations of an entity, including generics and ports. The Example 2 is presented here only for illustration purpose of the block syntax.

Important Notes

- Guarded blocks are generally not synthesizable.
- Unguarded blocks are usually ignored by synthesis tools.
- It is strongly recommended NOT to use blocks in non-VITAL designs - the package std_logic_1164 supports mechanisms and multiple value logic which make the reserved words bus, disconnect, guarded and register unnecessary. Also, instead of guarded blocks for modeling sequential behavior it is recommended to use clocked processes.
- VITAL specifications require the use of blocks.
- VHDL supports a more powerful mechanism of design partitioning which is called component instantiation. Component instantiation allows connecting a component reference in one entity with its declaration in another entity.

BOOLEAN

Definition

The BOOLEAN type is predefined in the STANDARD package as an enumerated data type with two possible values: FALSE and TRUE.

Syntax

```
type BOOLEAN is ( FALSE, TRUE );
```

Description

The BOOLEAN type is used for conditional operations. Boolean objects can be used with any of the relational operators =, /=, <, <=, >, >=.

According to the definition type, the leftmost value of the BOOLEAN type is FALSE, therefore the default value of any object of the BOOLEAN type is FALSE.

Since the BOOLEAN type is defined in the STANDARD package, it can be used in any VHDL specification without additional declarations.

Examples

Example 1

```
signal Condition_Signal : BOOLEAN;
-- ...
Condition_Signal <= TRUE;
-- ...
if Condition_Signal then -- could be: if Condition_Signal = true then
```

The Condition_Signal signal is declared as BOOLEAN but without any initial value. Therefore, by default it will be assigned the FALSE value. A conditional operation could have been used instead as shown in the comment, but such a form would contain useless redundancy and should be avoided.

Important Notes

- Unlike in traditional ("hand-based") digital design, BOOLEAN values (FALSE and TRUE) are NOT identical to logical '0' and '1', respectively. In VHDL, the latter form is a completely different type and is called the BIT type.

Case Statement

Formal Definition

The case statement selects for execution one of several alternative sequences of statements; the alternative is chosen based on the value of the associated expression.

Complete description: Language Reference Manual IEEE 1076-1993 section § 8.8.

Simplified Syntax

```
case expression is
  when choice => sequential_statements
  when choice => sequential_statements
  -- ...
end case;
```

Description

The case statement evaluates the listed expressions and selects one alternative sequence of statements according to the expression value. The expression can be of a discrete type or a one-dimensional array of characters (Example 1).

The case statement contains a list of alternatives starting with the `when` reserved word, followed by one or more choices and a sequence of statements.

An alternative may contain several choices (Example 2), which must be of the same type as the expression appearing in the case statement. For each expression there should be at least one locally static choice. The values of each choice must be unique (no duplication of values is allowed).

A choice can be either a simple name (Example 1), a name of a simple element (Example 2) or discrete range (a slice, Example 3). The choice types can be mixed.

A subtype with a constraint range (Example 4) can substitute a slice.

Another option is to use an object name as the choice. The object must be of the same type as the expression in the case statement. Example 5 shows it for a constant.

When all explicitly listed choices do not cover all the alternatives (all the values available for an expression of given type) the `others` choice must be used because the choice statements must cover all the alternatives, see Example 5).

Examples

Example 1

```
P1: process
  variable x : INTEGER range 1 to 3;
  variable y : BIT_VECTOR( 0 to 1 );
begin
  C1: case x is
    when 1 => Out_1 <= 0;
    when 2 => Out_1 <= 1;
    when 3 => Out_1 <= 2;
  end case C1;

  C2: case y is
    when "00" => Out_2 <= 0;
    when "01" => Out_2 <= 1;
    when "10" => Out_2 <= 2;
    when "11" => Out_2 <= 3;
  end case;
end process;
```

Depending on the values of the variable `x` and `y`, we assign the values 0, 1, 2 or 3 (in the second case) to the signals `Out_1` and `Out_2` (both of type `INTEGER`).

Example 2

```
P2: process
  type Codes_Of_Operation is ( ADD, UB, ULT, IV );
  variable Code_Variable : Codes_Of_Operation;
begin
  C3: case Code_Variable is
    when ADD | SUB => Operation := 0;
    when MULT | DIV => Operation := 1;
  end case;
end process;
```

When two or more alternatives lead to the same sequence of operations then they can be specified as a multiple choice in one `when` clause.

Example 3

```
P3: process
  type Some_Characters is ( 'a', 'b', 'c', 'd', 'e' );
  variable Some_Characters_Variable : Some_Characters;
begin
  C4: case Some_Characters_Variable is
    when 'a' to 'c' => Operation := 0;
    when 'd' to 'e' => Operation := 1;
  end case;
end process;
```

Slices can be used as choices. In such a case, the slice name must come from the discrete range of the expression type.

Example 4

```
P5: process
  variable Code_of_Operation : INTEGER range 0 to 2;
  constant Variable_1 : INTEGER := 0;
begin
  C6: case Code_of_Operation is
    when Variable_1 | Variable_1 + 1
      => Operation := 0;
    when Variable_1 + 2
      => Operation := 1;
  end case;
end process;
```

Constant used as a choice.

Example 5

```
P6: process
  type Some_Characters is ( 'a', 'b', 'c', 'd', 'e' );
  variable Code_of_Address : Some_Characters;
begin
  C7: case Code_of_Address is
    when 'a' | 'c' => Operation := 0;
    when others => Operation := 1;
  end case;
end process;
```

If the `Code_of_Address` variable is equal to 'a' or 'c', then the assignment `Operation := 0;` will be chosen. For the 'b', 'd' or 'e' values, the assignment `Operation := 1;` will be performed.

Important Notes

- The case expression must be of a discrete type or of a one-dimensional array type, whose element type is a `CHARACTER` type.
- Every possible value of the case expression must be covered by the specified alternatives; moreover, every value may appear only once (no duplicates or overlapping of ranges is allowed).
- The `when others` clause may appear only once and only as the very last choice.

CHARACTER

Formal Definition

The `CHARACTER` type object is an enumeration type object with at least one literal character among its enumeration literals.

Complete description: Language Reference Manual IEEE 1076-1993 § 3.1.1.1.

Syntax

Type `CHARACTER` is an enumeration type with all acceptable characters listed. Its specification can be found in the STANDARD package.

Description

The `CHARACTER` type is a predefined type declared in the STANDARD package and the values of this type are the 256 characters of the ISO 8859-1 (Latin 1).

The set of predefined operations for the `CHARACTER` type contains all relational functions: "=", "/=", "<', "<=", ">", ">=".

Single-line Comment

Formal Definition

A single-line comment starts with two adjacent hyphens and extends up to the end of the line.

Complete description: IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1993 § 13.8.

Syntax

```
comment ::= -- comment_string \n
```

Description

A comment starts with the double hyphen and ends with a new line character. If you need to comment out more than one line of a source code or your commentary does not fit in one row, the use of the single-line comments requires each line to start with the double hyphen character (--). Alternatively, you may use delimited comments available in VHDL 2008.

Comments may contain characters that are non-printable according to the Language Reference Manual.

Examples

Example 1

```
-- Copyright (c) Aldec, Inc.
-- All rights reserved.
```

Component Declaration

Formal Definition

A component declaration declares a virtual design entity interface that may be used in component instantiation statement.

Complete description: Language Reference Manual IEEE 1076-1993 § 4.5.

Simplified Syntax

```
component component_name [is]
  generic( generic_list );
  port( port_list );
end component [component_name];
```

Description

A component represents an entity/architecture pair. It specifies a subsystem, which can be instantiated in another architecture leading to a hierarchical specification. Component instantiation is like plugging a hardware component into a socket in a board (Fig. 1 in Example 1).

A component must be declared before it is instantiated. The component declaration defines the virtual interface of the instantiated design entity ("the socket") but it does not directly indicate the design entity.

The binding of a design entity to a given component may be delayed and may be placed either in the configuration specification or configuration declaration.

The component can be defined in a package, design entity, architecture, or block declarations. If the component is declared in an architecture, it must be declared before the `begin` statement of the architecture. In such a case, the component can be used (instantiated) in the architecture only.

A more universal approach is to declare a component in the package. Such a component is visible in any architecture, which uses this package.

Generics and ports of a component are copies of generics and ports of the entity the component represents.

Examples

Example 1

```
architecture STRUCTURE_2 of EXAMPLE is
  component XOR_4 is
    port(
      A : in BIT_VECTOR( 0 to 3 );
      B : in BIT_VECTOR( 0 to 3 );
      C: out BIT_VECTOR( 0 to 3 )
    );
  end component XOR_4;

  signal S1 : BIT_VECTOR( 0 to 3 );
  signal S2 : BIT_VECTOR( 0 to 3 );
  signal S3 : BIT_VECTOR( 0 to 3 );
begin
  X1: XOR_4 port map(
    S1 => A,
    S2 => B,
    S3 => C
  );
end architecture;
```

The XOR_4 component has two 4-bit input ports (A and B) and the 4-bit output port C. The declaration of this component is located in the declaration part of the architecture body STRUCTURE_2. The component instantiation statement assigns the X1 label to instantiated XOR_4 component and it associates its input-output interface with the S1, S2 and S3 signals.

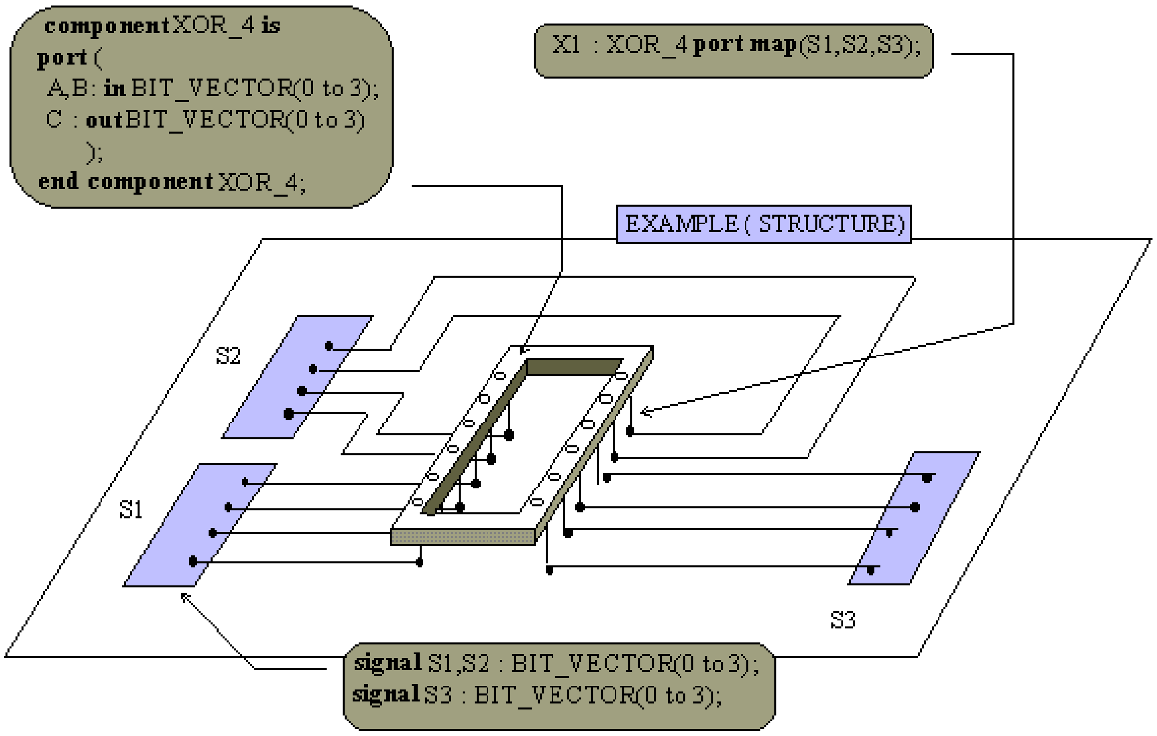

component XOR_4 is
port (
A,B: in BIT_VECTOR(0 to 3);
C : out BIT_VECTOR(0 to 3)
);
end component XOR_4;
X1 : XOR_4 port map(S1,S2,S3);
EXAMPLE (STRUCTURE)
S2
S1
S3
signal S1,S2 : BIT_VECTOR(0 to 3);
signal S3 : BIT_VECTOR(0 to 3);

Figure 1. Example of component declaration and instantiation

Important Notes

- A component declaration does not define which entity/architecture pair is bound to each instance. Such an information is defined by configuration.

Component Instantiation

Formal Definition

A component instantiation statement defines a subcomponent of the design entity in which it appears, associates signals or values with the ports of that subcomponent, and associates values with generics of that subcomponent.

Complete description: Language Reference Manual IEEE 1076-1993 § 9.6.

Simplified Syntax

```
label : [ component ] component_name
  generic map( generic_association_list )
  port map ( port_association_list );

label : entity entity_name [( architecture_identifier )]
  generic map( generic_association_list )
  port map ( port_association_list );

label : configuration configuration_name
  generic map( generic_association_list )
  port map ( port_association_list );
```

Description

A component represents an entity/architecture pair. It specifies a subsystem, which can be instantiated in another architecture, leading to a hierarchical specification. Component instantiation is like plugging a hardware component into a socket in a board (Fig. 1 in Example 1).

The component instantiation statement introduces a subsystem declared elsewhere, either as a component or as an entity/architecture pair (without declaring it as a component).

The component instantiation contains a reference to the instantiated unit and actual values for generics and ports. There are three forms of component instantiation:

- instantiation of a component;
- instantiation of a design entity;
- instantiation of a configuration;

See configuration for details on the third form.

The actual values of generic map aspect and port map aspect connections allow assigning the components of the actual values to generic parameters and ports.

Instantiation of a Component

Instantiation of a component introduces a relationship to a unit defined earlier as a component (see component declaration). The name of the instantiated component must match the name of the declared component. The instantiated component is called with the actual parameters for generics and ports. The association list can be either positional or named.

In the positional association list, the actual parameters (generics and ports) are connected in the same order in which ports were declared in the component (Example 1).

Named association allows to list the generics and ports in an order that is different from the one declared for the component. In such a case the ports have to be explicitly referenced (Example 2).

Instantiation of a Design Entity

It is not necessary to define a component to instantiate it: the entity/ architecture pair can be instantiated directly. In such a direct instantiation, the component instantiation statement contains the design entity name and optionally the name of the architecture to be used for this design entity. The reserved word `entity` follows the declaration of this type of the component instantiation statement (Example 3).

If architecture name is not specified in an instantiation of a design entity, the last compiled architecture associated with the entity will be taken.

Examples

Example 1

```
architecture Structural of ALU is
  signal X, Y, S, C : BIT;
  component Half_Adder is
    port(
      In1 : in BIT;
      In2 : in BIT;
      Sum : out BIT;
      Carry : out BIT
    );
  end component;
begin
  HA: half_adder port map ( X, Y, S, C );
  -- ...
end architecture;
```

The structural specification of an arithmetic-logic unit `ALU` uses an instantiation of a `Half_Adder` component. Note that the component is instantiated with signals of the `ALU` system. The signals are associated positionally.

Example 2

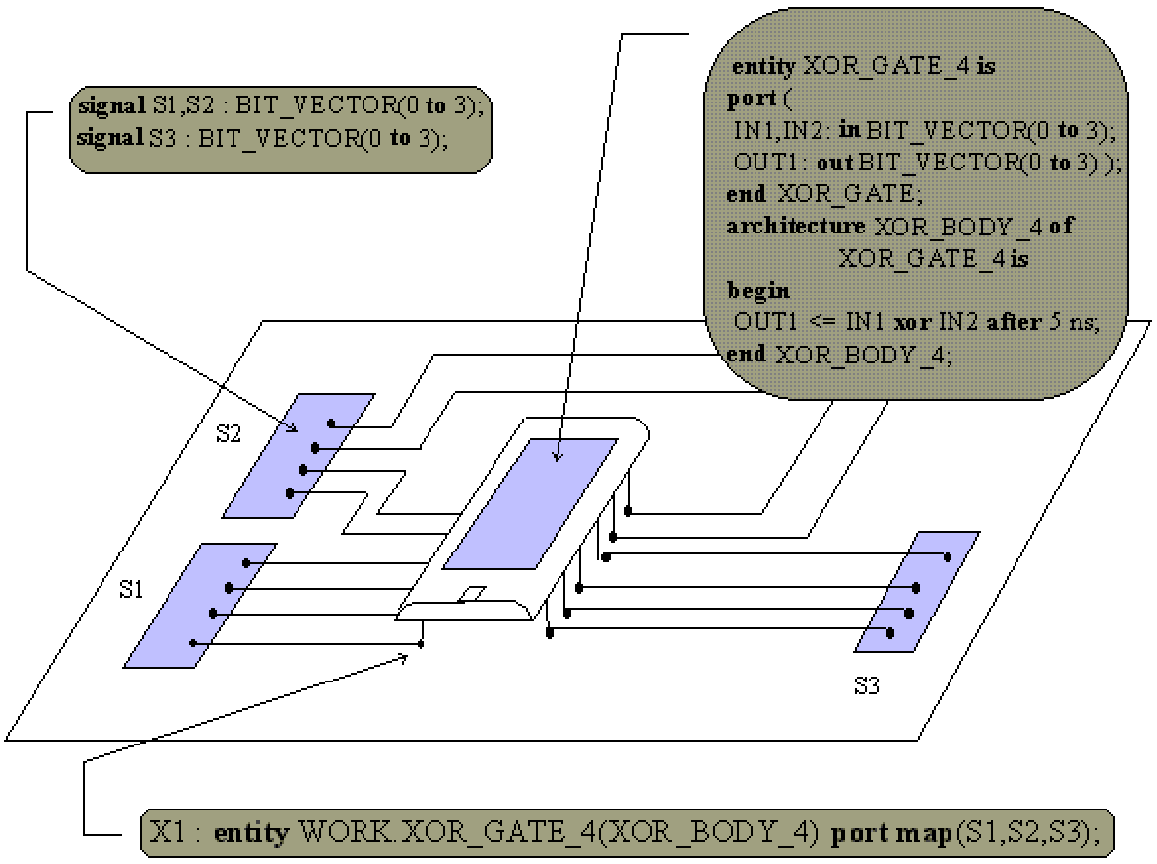
```
architecture Structural of ALU is
  signal X, Y, S, C : BIT;
  component Half_Adder is
    port(
      In1 : in BIT;
      In2 : in BIT;
      Sum : out BIT;
      Carry : out BIT
    );
  end component;
begin
  HA: half_adder port map(
    Sum => S,
    Carry => C,
    In1 => X,
    In2 => Y
  );
  -- ...
end architecture;
```

This structural architecture performs the same function as in the Example 1. The only difference lies in the way the association list is specified for the component ports - the signals are associated with named association.

Example 3

```
entity XOR_GATE_4 is
  port(
    IN1 : in BIT_VECTOR( 0 to 3 );
    IN2 : in BIT_VECTOR( 0 to 3 );
    OUT1 : out BIT_VECTOR( 0 to 3 )
  );
end entity;

architecture XOR_BODY_4 of XOR_GATE_4 is
begin
  OUT1 <= IN1 xor IN2;
end architecture;

entity EXAMPLE is
end entity;

architecture STRUCTURE_1 of EXAMPLE is
  signal S1, S2, S3 : BIT_VECTOR( 0 to 3 );
begin
  X1: entity work.xor_gate_4( xor_body_4 )
    port map( S1, S2, S3 );
end architecture;
```

Entity `XOR_GATE_4` is directly instantiated here, without declaring a component. The architecture, which specifies the body of the entity `XOR_GATE_4` is called `XOR_BODY_4` and is supported in parentheses. Further specification is similar to the one in instantiation of a component. The entity and architecture instantiated here must be located in the WORK library prior to this instantiation.



```
signal S1,S2 : BIT_VECTOR(0 to 3);
signal S3 : BIT_VECTOR(0 to 3);
```

```
entity XOR_GATE_4 is
port (
 IN1,IN2: in BIT_VECTOR(0 to 3);
 OUT1 : out BIT_VECTOR(0 to 3) );
end XOR_GATE;
architecture XOR_BODY_4 of
           XOR_GATE_4 is
begin
 OUT1 <= IN1 xor IN2 after 5 ns;
end XOR_BODY_4;
```

```
X1 : entity WORK.XOR_GATE_4(XOR_BODY_4) port map(S1,S2,S3);
```

Figure 1. Example of a direct instantiation.

Important Notes

- The label for component instantiation is obligatory.

Composite Type

Formal Definition

A composite type object is one having multiple elements. There are two classes of composite types: array types and record types.

Complete description: Language Reference Manual IEEE 1076-1993 § 3, § 3.2.

Syntax

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

Description

An object of a composite type is a collection of other objects, called elements. The elements can be of any scalar, composite or access type. It is not allowed to use file types as elements of a composite type.

The difference between arrays and records lies in that all elements of an array must be of the same type. For example, each array element can be a voltage value. On the other hand, each element of a record can be of a different type (Voltage1, current1, resistance1, Voltage2, ...). Each element of an array is referred to by its array name and position (index). On the other hand, the record elements (called fields) are referred to through their individual names (together with the name of entire record), or through an aggregate.

See Array and Record for details.

Examples

Example 1

```
type T_Monthly_Income is array ( 1 to 12 ) of INTEGER;
type T_Personal_Data is
  record
    First_Name : STRING( 1 to 6 );
    Last_Name : STRING( 1 to 10 );
    ID_Number : STRING( 1 to 5 );
    Incomes : T_Monthly_Income;
    Tax_Paid : BOOLEAN;
  end record;
```

The two types above illustrate two classes of composite type. The first one defines a 12-element array of integer values, while the second one is a record with five fields. First four fields are of composite types, while the very last one is of a scalar type.

Example 2

```
signal Year_97_Income : T_Monthly_Income;
signal Some_Employee : T_Personal_Data;
  -- ...
Year_97_Inc(12) <= 5500;
Some_Employee.First_Name <= "Gordon";
```

The signals are declared to be of the types declared in Example 1. Note the way elements of the composite types are accessed: in case of arrays through the position index, while in record through the field name (preceded by a dot).

Concatenation

Formal Definition

Predefined adding operator for any one-dimensional array type.

Complete description: Language Reference Manual IEEE 1076-1993 section § 7.2.4.

Description

The concatenation operator (denoted as &) composes two one-dimensional arrays into a larger one of the same type. A single element can be used as any of the two operands of concatenation. If two single elements are concatenated, then the result can be of any array type (as long as it is compatible with the type of the operands).

The resulting array is composed of the elements of the left operand (in left-to-right order) followed by the elements of the right operand (in the same order). The direction of the resulting array is the same as of the left operand, unless the left operand is a null array, in which case the direction of the result is that of the right operand.

Examples

```
variable Byte_Data : BIT_VECTOR( 7 downto 0 );
alias Sign : BIT is Byte_Data( 7 );
alias Modulus : BIT_VECTOR( 6 downto 0 ) is Byte_Data( 6 downto 0 );
constant Four_Zeros : BIT_VECTOR( 3 downto 0 ) := ( others => '0' );
constant Reset_High : BIT_VECTOR( 7 downto 0 ) := Four_Zeros & "1111";
constant Reset_All : BIT_VECTOR( 7 downto 0 ) := Four_Zeros & Four_Zeros;
Byte_Data := '1' & Modulus;
```

Important Notes

- The declared number of elements in the result array must be large enough to cover the number of both operands added.

Configuration Declaration

Formal Definition

A configuration is a construct that defines how component instances in a given block are bound to design entities in order to describe how design entities are put together to form a complete design.

Complete description: Language Reference Manual IEEE 1076-1993 § 1.3

Simplified Syntax

```
configuration configuration_name of entity_name is
  -- configuration declarations
  for architecture_name
    for instance_label:component_name
      use entity library_name.entity_name( arch_name );
    end for;
    -- other for clauses
  end for;
end [configuration] [configuration_name];

configuration configuration_name of entity_name is
  -- configuration declarations
  for architecture_name
    for instance_label:component_name
      use configuration library_name.config_name;
    end for;
    -- other for clauses
  end for;
end [configuration] [configuration_name];
```

Description

Each component instantiation refers to some design entity (entity/architecture pair) and the association is specified by a configuration specification. Component specification appears in the declarative part of the unit, where the instances are used. If for some reasons, however, it is appropriate (or desired) to postpone (or defer) such association until later, configuration declaration can be used for specifying such deferred component specifications.

The configuration declaration starts with the configuration name and then it is associated to a given design entity. Declarative part of the configuration may contain use clauses, attribute specifications and group declarations. The main part of the configuration declaration contains so called block configuration. It indicates which architecture will be used with the entity specified earlier, as well as which configuration elements will be used in the component instantiation. This declaration may contain other blocks' configurations, allowing this way to specify hierarchical structures. Such a configuration can be called hierarchical, while a configuration without hierarchy can be called simple.

A simple configuration contains reference to only one architecture body (Example 1).

Hierarchical configurations, on the other hand, allow to nest block configurations (Example 2). This mechanism allows binding component instantiation statements with the design entities down the hierarchy.

When the ports and generics in component declaration do not match with their counterparts in entity declaration, so called binding indication can be applied. Simply speaking this is an explicit notification on how the ports and generics in the entity should be bound to ports and generics of the component instance. The `generic map` and `port map` clauses are used for this purpose. This technique is used in Example 1. In practice, however, it is recommended to match the generics and ports of components and respective entities as this improves readability.

Two main binding methods that can be applied in configuration specifications: binding of entities and binding configurations. They are illustrated in Example 1 and Example 3, respectively.

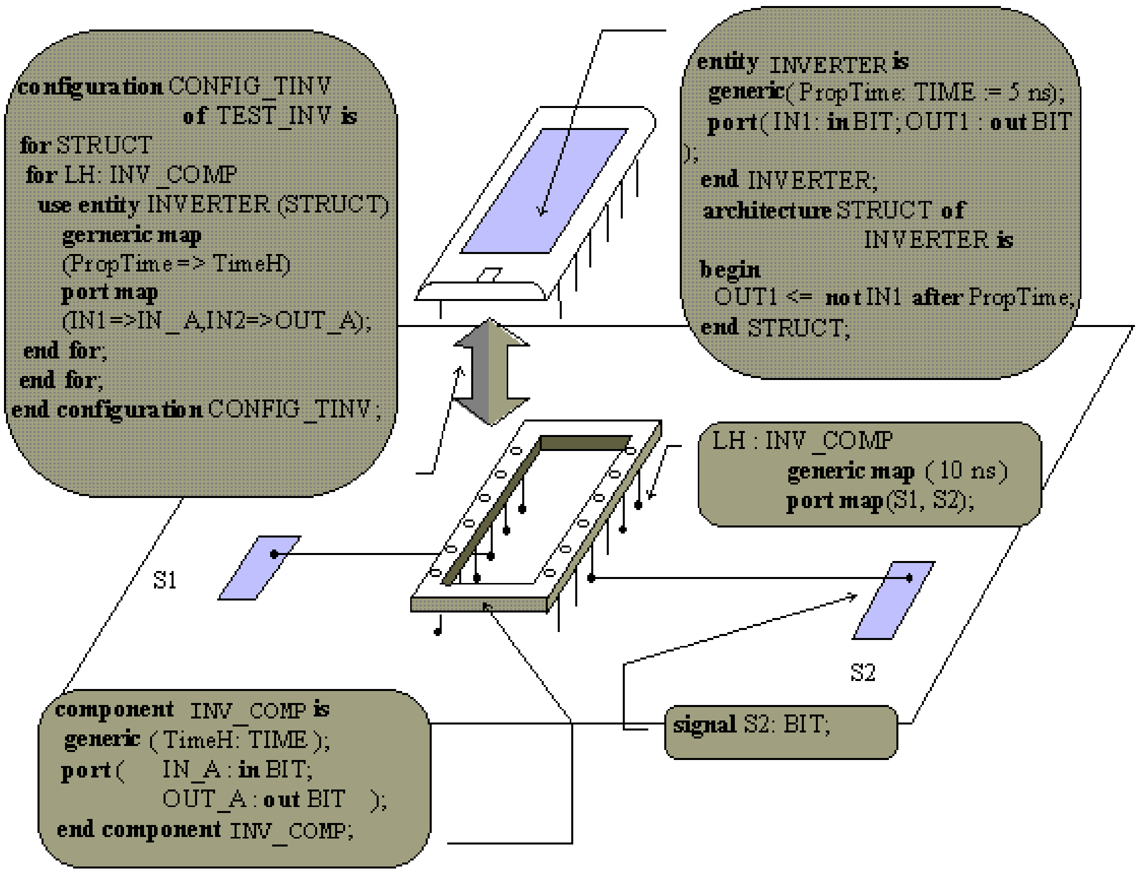
Examples

Example 1

```
entity INVERTER is
  generic(
    Propagation_Time : TIME := 5 ns );
  port(
    IN1 : in BIT;
    OUT1 : out BIT );
end entity;

architecture STRUCT_I of INVERTER is
begin
  OUT1 <= not IN1 after Propagation_TIME;
end architecture;

entity TEST_INV is
end entity;

architecture STRUCT_T of TEST_INV is
  signal S1, S2 : BIT := '1';
-- INV_COMP component declaration:
  component INV_COMP is
    generic(
      Time_High : TIME );
    port(
      IN_A : in BIT;
      OUT_A : out BIT );
  end component;
begin
  -- instantiation of INV_COMP component:
  LH: INV_COMP
    generic map( 10 ns )
    port map( S1, S2 );
end architecture;

configuration CONFIG_TEST_INV of TEST_INV is
  for STRUCT_T -- indicates architecture body of TEST_INV
    -- indicates design entity for LH instantiation statement:
    for LH: INV_COMP
      use entity WORK.INVERTER( STRUCT_I )
      -- indicates generic and port aspects:
        generic map( Propagation_Time => Time_High )
        port map( IN1 => IN_A, OUT1 => OUT_A );
    end for;
  end for;
end configuration;
```

The CONFIG_TEST_INV configuration declaration can be used as an example of the basic configuration declaration. There is only one block configuration in the configuration declaration. This block contains a component declaration INV_COMP. In the component instantiation statement LH, the design entity INVERTER is assigned to INV_COMP component.

There is one block configuration in the CONFIG_TEST_INV configuration declaration, it indicates that STRUCT_T architecture body will be used. INV_COMP component configuration appears in the block configuration. The CONFIG_TEST_INV configuration for the TEST_INV design entity allows associating LG component: INV_COMP with INVERTER design entity and its STRUCT_1 architecture body.



```
configuration CONFIG_TINV
        of TEST_INV is
 for STRUCT
  for LH: INV_COMP
   use entity INVERTER (STRUCT)
     gerneric map
     (PropTime => TimeH)
     port map
     (IN1=>IN_A,IN2=>OUT_A);
  end for;
 end for;
end configuration CONFIG_TINV;
```

```
entity INVERTER is
 generic( PropTime: TIME := 5 ns);
 port( IN1: in BIT; OUT1 : out BIT
);
 end INVERTER;
 architecture STRUCT of
            INVERTER is
 begin
  OUT1 <= not IN1 after PropTime;
 end STRUCT;
```

```
LH : INV_COMP
     generic map ( 10 ns)
     port map(S1, S2);
```

```
component  INV_COMP is
 generic ( TimeH: TIME );
 port (    IN_A : in BIT;
           OUT_A : out BIT   );
end component INV_COMP;
```

```
signal S2: BIT;
```

Figure 1. Example of configuration declaration

Example 2

```
-- block configuration for architecture body STRUCT:
for STRUCT
-- component configuration specified in architecture body STRUCT:
  for SPEECH_CPU: SPEECH
    use entity SP_LIB.DIG_REC( FAST )
      generic map( Time_Record => 20 sec );
    -- block configuration for architecture body FAST of DIG_REC:
    for FAST
      -- component configuration specified in architecture body FAST:
      for AD_CONV: ADC_1 -- relates to instance AD_CONV of ADC_1
        use entity ANALOG_DEV.ADC;
      end for; -- for AD_CONV
    end for; -- for FAST
  end for; -- for SPEECH_CPU
end for; -- for STRUCT
```

The block configuration, indicating architecture body STRUCT, appears in the configuration declaration. Next, in the block configuration there is the component configuration SPEECH, which also contains block configuration FAST. The block configuration FAST configures the architecture body FAST that contains an instantiation statement with label AD_CONV. The entire block is hierarchically specified here.

Example 3

```
configuration Conf_Test of Test is
  for STRUCTURE_T
    for T_1 : DEC use configuration CONF_E;
    end for;
  end for;
end configuration;
```

In this example, the configuration declaration of design entity EXAMPLE is used. It binds EXAMPLE design entity to DEC component by using its configuration CONF_E as an entity aspect for T_1 component instance in the body architecture STRUCTURE_T.

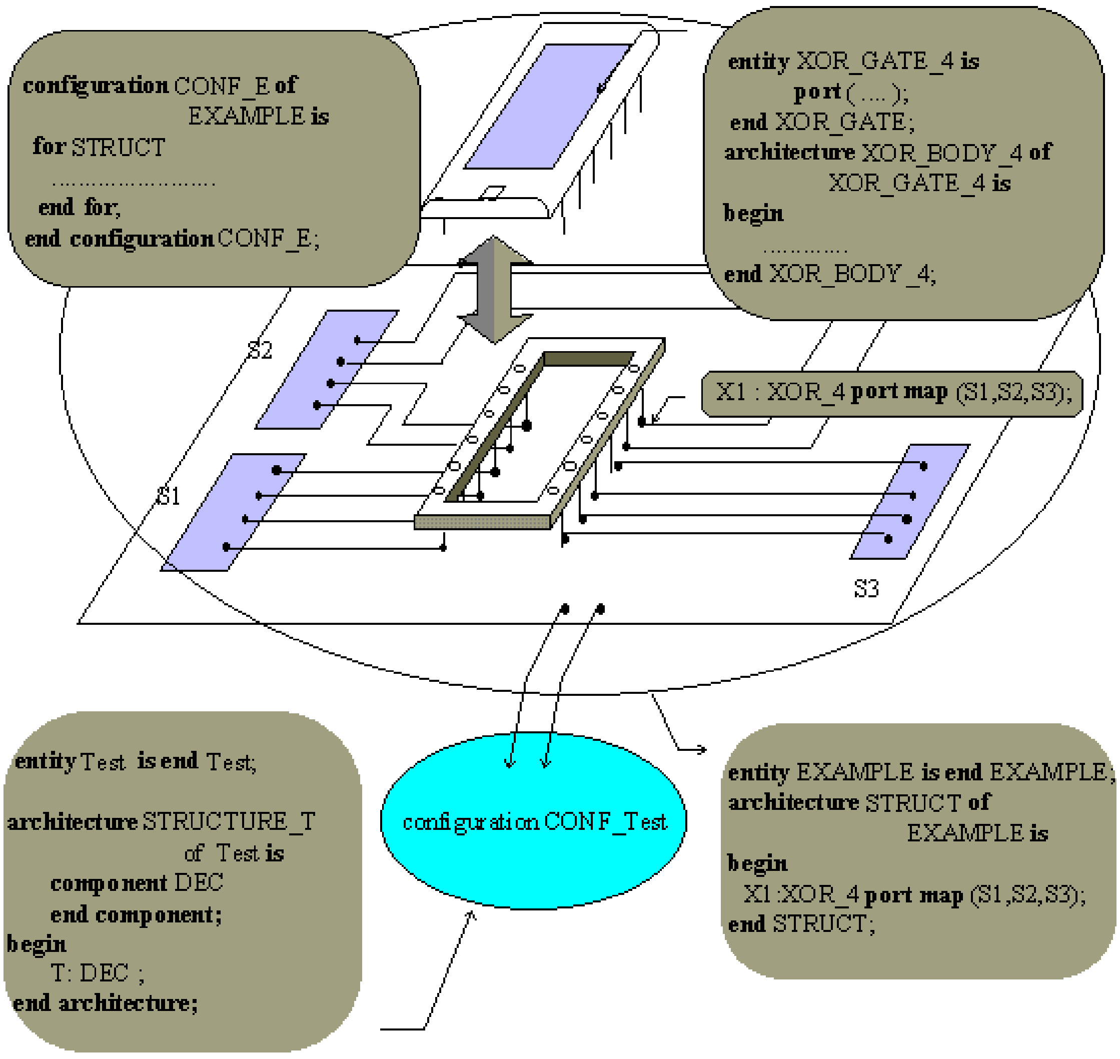

configuration CONF_E of EXAMPLE is
for STRUCT
....
end for;
end configuration CONF_E;
entity XOR_GATE_4 is
port (....);
end XOR_GATE;
architecture XOR_BODY_4 of XOR_GATE_4 is
begin
....
end XOR_BODY_4;
S2
S1
S3
X1 : XOR_4 port map (S1,S2,S3);
entity Test is end Test;
architecture STRUCTURE_T of Test is
component DEC
end component;
begin
T: DEC ;
end architecture;
configuration CONF_Test
entity EXAMPLE is end EXAMPLE;
architecture STRUCT of EXAMPLE is
begin
X1 :XOR_4 port map (S1,S2,S3);
end STRUCT;

Figure 2. Example of configuration declaration

Important Notes

- Configuration assigns one and only one architecture to a given entity.
- Synthesis tools do generally not support configurations.
- For a configuration of some design entity, both the entity and the configuration must be declared in the same library.

Configuration Specification

Formal Definition

A configuration is a construct that defines how component instances in a given block are bound to design entities in order to describe how design entities are put together to form a complete design.

Complete description: Language Reference Manual IEEE 1076-1993 § 5.2.

Simplified Syntax

```
for instance_label:component_name
  use entity library_name.entity_name( arch_name );

for instance_label:component_name
  use configuration library_name.config_name;
```

Description

Each component instantiation refers to some design entity (entity/architecture pair) and the association is specified by a configuration specification. Component specification appears in the declarative part of the unit, where the instances are used. This way components can be configured within architecture which instances them without using a separate configuration declaration. The specification is simpler, but also less flexible. Example 1 contains a configuration specification for the same component as in the Example 1 in the configuration declaration description.

When the ports and generics in component declaration do not match with their counterparts in entity declaration, so called binding indication can be applied. Simply speaking this is an explicit notification on how the ports and generics in the entity should be bound to ports and generics of the component instance. The `generic map` and `port map` clauses are used for this purpose. This technique is used in Example 1. In practice, however, it is recommended to match the generics and ports of components and respective entities as this improves readability.

If no configuration (either in the form of a declaration or specification) is supported for a component, so called default binding will occur. This means that for such a component an entity will be selected such that its name, port names, port types, generics etc. match those in the corresponding component declaration. If the entity has more than one architecture, the last analyzed of them will be used.

Examples

Example 1

```
entity INVERTER is
  generic(
    Propagation_Time : TIME := 5 ns );
  port(
    IN1 : in BIT;
    OUT1 : out BIT );
```

```
end entity;

architecture STRUCT_I of INVERTER is
begin
  OUT1 <= not IN1 after Propagation_TIME;
end architecture;

entity TEST_INV is
end entity;

architecture STRUCT_T of TEST_INV is
  signal S1, S2 : BIT := '1';
  -- INV_COMP component declaration:
  component INV_COMP is
    generic(
      Time_High : TIME );
    port(
      IN_A : in BIT;
      OUT_A : out BIT );
  end component;

  for LH : INV_COMP
    use entity INVERTER ( STRUCT_I )
  -- indicates generic and port aspects:
      generic map(
        Propagation_Time => Time_High )
      port map(
        IN1 => IN_A,
        OUT1 => OUT_A );
begin
  -- instantiation of INV_COMP component:
  LH : INV_COMP
    generic map( 10 ns )
    port map( S1, S2 );
end architecture;
```

Architecture STRUCT_T of the entity TEST_INV uses a component INV_COMP. The binding of the component to the entity INVERTER and architecture STRUCT_I is implemented by the configuration specification that appears in the declarative part of the architecture.

Important Notes

- Synthesis tools do generally not support configurations. Users are required to ensure that component and entity names, ports and generics match (default binding).
- For a configuration of some design entity, both the entity and the configuration must be declared in the same library.

Constant

Formal Definition

Constant is an object whose value cannot be changed once defined for the design. Constants may be explicitly declared or they may be sub-elements of explicitly declared constants, or interface constants. Constants declared in packages may also be deferred constants.

Complete description: Language Reference Manual IEEE 1076-1993 § 4.3.1.1.

Simplified Syntax

```
constant constant_name : type := value;
```

Description

A constant is an object whose value may never be changed during the simulation process.

The constant declaration contains one or more identifiers, a subtype indication and an expression which specifies the value of the constant declared in the particular statement. The identifiers specify names of the constants. Each name appearing in the identifier list creates a separate object.

The object type in the constant declaration can be of scalar or composite type and it can be constrained. A constant cannot be of the file or access type. If a constant is an array or a record then none of its elements can be of the file or access type.

The expression used in the constant declaration must refer to a value of the same type as specified for the constant (Example 1).

If a constant is declared an array other than `STRING`, `BIT_VECTOR` or `STD_LOGIC_VECTOR`, then the value for the constant must be specified using aggregates (Example 2).

A constant declared in a package can be deferred, i.e. it can be declared without specifying its value, which is given later on, in the package body (Example 3).

constants> improve the clarity and readability of a project. Moreover, they simplify incorporating changes in the project. For example, if a design contains a bus with a fixed width, a constant representing the number of bits in the bus can be used. When the width of the bus is to be changed, it is sufficient to alter the constant declaration only.

The visibility of constants> depends on the place of their declaration. The constants> defined in the package can be used by several design units. The constant declaration in the design entity is seen by all the statements of the architecture bodies of this entity. The constants >defined in the declaration part of the design unit is seen in all bodies related to this design, including the process statement. The constant defined in the process can only be used in this process.

Examples

Example 1

```
type Week_Day is ( Mon, Tue, Wed, Thu, Fri, Sat, Sun );
constant Start_Day : Week_Day := Sat;
constant Logical_GND : Bit := '0';
constant Bus_Width, Queue_Length : INTEGER := 16;
constant Clock_Period : TIME := 15 ns;
constant Max_Simulation_Time : TIME := 200 * Clock_Period;
```

Each of the six constants above is of a scalar type. Both `Bus_Width` and `Queue_Length` are expected to be INTEGER numbers of the same value, therefore they were specified using one declaration. Note that you can either explicitly specify the constant's value or using an expression based on other constants (see the `Max_Simulation_Time` constant).

Example 2

```
type Numeric_Code_Type is array ( 7 downto 0 ) of INTEGER range 0 to 9;
constant Entry_Code : Numeric_Code_Type := ( 2, 6, 4, 8, 0, 0, 1, 3 );
constant Data_Bus_Reset: STD_LOGIC_VECTOR( 7 downto 0 ) := ( others => '0' );
```

Both constants are of complex types, but `Data_Bus_Reset` is of the `STD_LOGIC_VECTOR`, thus it can be assigned its value directly. `Entry_Code` is also a one-dimensional array, but its elements are integers so the value for the constant must be specified using an aggregate numeric code type (2, 6, 4, 8 etc.).

Example 3

```
package Timing is
  constant Reset : STD_LOGIC;
end package;

package body Timing is
  constant Reset: STD_LOGIC := '0';
end package body;
```

Note that the `Reset` constant is declared in the package without a concrete value assigned to it because the complete declaration of this constant is given in the package body.

Important Notes

- By definition, a constant may not be assigned any values by the simulation process.
- Use constants as often as possible as they create more readable and maintainable code.
- Use constants to define data parameters and lookup tables, which may substitute function calls the simulation time of such lookups is significantly shorter than that of function calls.

Delay

Formal Definition

Delay is a mechanism allowing introducing timing parameters of specified systems.

Complete description: Language Reference Manual IEEE 1076-1993 § 8.4.

Syntax

```
delay_mechanism ::= transport | [ reject time_expression ] inertial
```

Description

The delay mechanism allows introducing propagation times of described systems. Delays are specified in signal assignment statements. It is not allowed to specify delays in variable assignments.

There are two delay mechanism available in VHDL: inertial delay (default) and transport delay.

The transport delay is defined using the reserved word `transport` and is characteristic for transmission lines. New signal value is assigned with specified delay independently from the width of the impulse in waveform (i.e. the signal is propagated through the line - Example 1).

Inertial delay is defined using the reserved word `inertial` and is used to model the devices, which are inherently inertial. In practice this means, that impulses shorter than specified switching time are not transmitted (Example 2).

Inertial delay specification may contain a `reject` clause. This clause can be used to specify the minimum impulse width that will be propagated, regardless of the switching time specified (Example 3).

If the delay mechanism is not specified then by default it is inertial.

Examples

Example 1

```
B_OUT <= transport B_IN after 1 ns;
```

The value of the signal `B_IN` is assigned to the signal `B_OUT` with 1 ns delay. The distance between subsequent changes of `B_IN` is not important - all changes are transmitted to `B_OUT` with specified delay (Fig. 1).

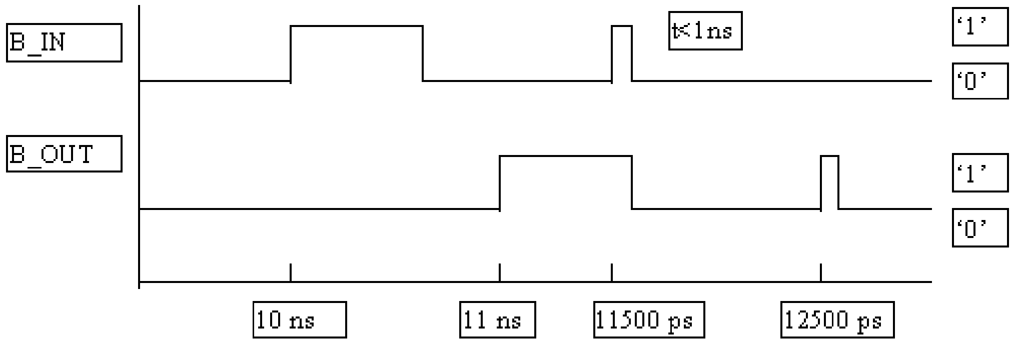


Figure 1. Example of transport delay

Example 2

```
L_OUT <= inertial L_IN after 1 ns;
```

The signal value `L_IN` is assigned to the signal `L_OUT` with 1 ns delay. Not all changes of the signal `L_IN`, however, will be transmitted: if the width of an impulse is shorter than 1 ns then it will not be transmitted. See Fig. 2 and the change of `L_IN` at 13 ns and again at 13.7 ns.

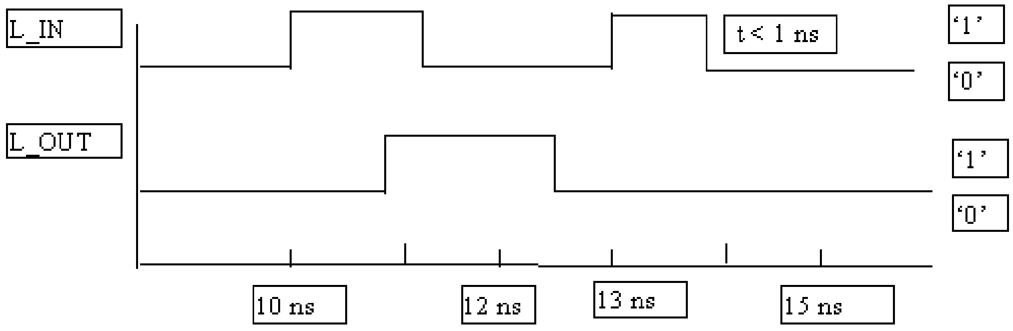


Figure 2. Example of inertial delay

Example 3

```
Q_OUT <= reject 500 ps inertial Q_IN after 1 ns;
```

The signal value `Q_IN` is assigned to the signal `Q_OUT` with 1 ns delay. Although it is an inertial delay with switching time equal to 1 ns, the reject time is specified to 500 ps (0.5 ns) and only impulses shorter than 500 ps will not be transmitted (Fig. 3).

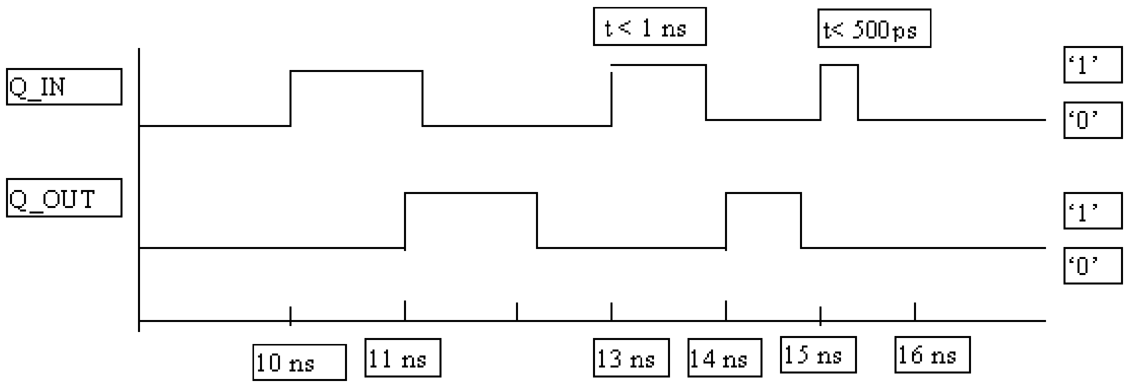


Figure 3. Example of inertial delay with rejection limit.

Important Notes

- Delay mechanisms can be applied to signals only. It is not allowed to specify delays in variable assignments.
- Delays are not synthesizable.
- The inertial delay is the default delay and the reserved word `inertial` can be omitted.

Driver

Formal Definition

A container for a projected output waveform of a signal. The value of the signal is a function of the current values of its drivers. Each process that assigns to a given signal implicitly contains a driver for that signal. A signal assignment statement affects only the associated driver(s).

Complete description: Language Reference Manual IEEE 1076-1993 § 12.6.1.

Description

Each signal assignment statement defines a driver for each scalar signal that is a target of this assignment. In case of signals of complex type, each element has its own driver. Inside processes each signal has only one driver, no matter how many assignment to it are specified.

When an assignment statement is executed, a new value is assigned to the signal driver. The value of the signal is determined based on all its drivers using the resolution function.

Examples

```
signal Data_Bus : STD_LOGIC_VECTOR( 7 downto 0 ) := ( others => 'Z' );
P1: process ( A, B )
begin
  -- ...
  Data_Bus <= ( others => '1' );
end process;

P2: process ( A, B )
begin
  -- ...
  Data_Bus <= ( others => '0' );
end process;
```

Signal `Data_Bus` is assigned values in two processes, therefore it will have two drivers (one per each process). The assignments will result in a change of the value of respective drivers, which will result in assigning the "XXXXXXXX" value to the `Data_Bus`.

Important Notes

- Drivers are not associated with signal declarations but with signal assignments.
- If a signal has more than one driver in an architecture, it must be of a resolved type.

Entity

Formal Definition

Entity is the description of the interface between a design and its external environment. It may also specify the declarations and statements that are part of the design entity. A given entity declaration may be shared by many design entities, each of which has a different architecture. Thus, an entity declaration can potentially represent a class of design entities, each having the same interface.

Complete description: Language Reference Manual IEEE 1076-1993 § 1, § 1.1.

Simplified Syntax

```
entity entity_name is
  [generic ( generic_list );]
  [port( port_list );]
[begin
  statements]
end [entity] [entity_name];
```

Description

An entity specifies the interface between the specified design (formally called a design entity) and the environment in which it operates. On the other hand, an architecture is a description of the inner design operation and it must be assigned to an entity. The architecture can be assigned to one entity only but one entity may be assigned to a number of architectures.

The entity statement declares the design name (the identifier item in the Syntax example). In addition, it defines generic parameters (see Generic) and ports (see Port) of the design entity. Generic parameters provide static information (like timing parameters or bus width) to a design. Ports provide communication channels between the design and its environment. For each port, its mode (i.e. data flow) and type are defined.

Optionally, an entity may contain a declarative part. Any subprograms, types, subtypes, and constants can be declared here.

Declarations which are defined in an entity are visible to all architectures assigned to this entity.

An entity may contain its own statements, declared after the `begin` keyword. The statements here must be passive, which means they cannot alter values of any signals; Passive processes, concurrent assertion statements and passive concurrent procedure calls can be used here.

The entity declaration may be preceded by the library and use clauses. This way all declarations defined in a package will be visible for the entity and all architectures assigned to it.

Examples

Example 1

```
library IEEE;
use IEEE.std_logic_1164.all;

entity BCD_Decoder is
port(
  BCD : in BIT_VECTOR( 2 downto 0 );
  Enable : in BIT;
  LED : out STD_ULOGIC_VECTOR( 3 downto 0 ) );
constant ZERO : STD_ULOGIC_VECTOR( 3 downto 0 ) := "0000";
begin
  assert ( BCD /= "111" ) report "BCD= 7 " severity note;
end entity BCD_Decoder;
```

The above example illustrates several important issues related to entities. First two lines contain a call to the IEEE library and to the std_logic_1164 package, respectively. These two lines are required because the STD_ULOGIC_VECTOR type used for the output signal LED is not a standard type but it is defined in the mentioned package. If LED would be of BIT_VECTOR type then the two lines could have been omitted.

The BCD_Decoder identifier, which is following the entity keyword, is a name assigned by the designer to the entity. Note that this name is repeated at the very end of the entity.

The above listed entity contains the specification of ports only. In this case there are two inputs (BCD and Enable) and one output (LED). The mode for each of them is supported after a colon and is followed by a specification of the signal's type. See Port for more details on modes and types of ports.

The declarative part of the above entity contains two declarations: constant and assert statements. The constant introduced here will be visible in all architectures of the BCD_Decoder entity. This type of a declaration makes sense if there are more than one such architectures. Otherwise, it might be better to place it in the architecture section to make the entity more clear. See Constant for more details on constants.

The assert statement is a concurrent statement which will be active whenever any of the BCD_Decoder architectures is active. This particular statement will generate a message listed in the report clause, whenever BCD will be equal to "111" ("BCD = 7"). Note that the condition in the assert statement should be interpreted as "if not condition - then report". Turn to Assertion Statement for more information on this topic.

Important Notes

- The VHDL Language Reference Manual uses the name design entity for a complete specification of the design, i.e. both its interface (entity unit) and behavior or structure (architecture unit). Therefore entity and design entity are not the same concepts!
- The identifier for an entity must conform to VHDL identifier rules; it must start with a letter followed by an arbitrary combination of letters, digits and underline symbols.
- While it is not necessary to repeat the name of an entity at the end of the declaration, it is strongly recommended to do it for the sake of clarity of the description; for the same reason it is advised to add the entity keyword between the end and the entity name.
- It is possible to write an entity without any generics, ports and passive statements. In fact this is used in constructing testbenches (see Testbench).

Enumeration Type

Formal Definition

An enumeration type is a type whose values are defined by listing (enumerating) them explicitly. This type values are represented by enumeration literals (either identifiers or character literals).

Complete description: Language Reference Manual IEEE 1076-1993 § 3.1.1.

Syntax

```
type type_name is ( enumeration_literal, enumeration_literal, ... );
```

Description

The enumeration type is a type with an ordered set of values, called enumeration literals, and consisting of identifiers and character literals. Each of the enumeration literals must be unique within the given declaration type, but different enumeration types may use the same literals (Example 1). In this case, it is said that such literals are overloaded. When such a literal is referenced in the source code, its is determined from the context, in which enumeration this literal has occurred.

All enumerated values are ordered and each of them has a numeric (integer) value assigned to it. The number indicates the position of the literal. The very first literal in the definition has position number zero and each subsequent has the number increased by one from its predecessor (Example 2).

Each enumeration type defined has implicitly defined relational operators that can be used on the type values.

The package Standard contains declarations of several predefined enumeration types: BIT, BOOLEAN, CHARACTER, SEVERITY_LEVEL, FILE_OPEN_KIND and FILE_OPEN_STATUS. Apart from that the package std_logic_1164 defines another enumeration type, STD_ULOGIC.

Examples

Example 1

```
type Not_Good is ( X, '0', '1', X ); -- illegal
type My_Bit is ( L, H );
type Test is ( '0', '1', L, H );
```

The type `Not_Good` is an illegal declaration as the literal `X` appears twice in the same declaration. On the other hand there is nothing incorrect in using `L` (low) and `H` (high) twice because they are used in two different declarations.

Example 2

```
type FSM_States is ( Init, Read, Decode, Execute, Write );
```

The type FSM_States defines five possible values, which are numbered from 0 to 4: the position number of Init is 0, position of Read is 1, Decode - 2, Execute - 3, and Write - 4.

Important Notes

- It is illegal to define an enumeration type with a range.
- It is assumed that the values are defined in ascending order. For this reason it is recommended to order the literals in such a way that the default value is the first one (it is referred to through the attribute 'left').
- Objects of enumeration types are typically synthesizable.

Event

Formal Definition

A change in the current value of a signal, which occurs when the signal is updated with its effective value.

Complete description: Language Reference Manual IEEE 1076-1993 § 12.6.2.

Description

The event is an important concept in VHDL. It relates to signals and it occurs on a signal if the current value of that signal changes. In other words, an event on a signal is a change of the signal's value.

It is possible to check whether an event occurred on a signal. Such an information can be obtained through the predefined attribute 'EVENT. The principal application of this attribute is checking for an edge of a clock signal (Example 1). It is also possible to check when the last event on a signal occurred (attribute 'LAST_EVENT). See attributes for details.

An event on a signal, which is on sensitivity list of a process or is a part of an expression in a concurrent signal assignment, causes the process or assignment to resume (invoke). See Sensitivity List and Resume for details.

Examples

Example 1

```
if CLK = '1' and CLK'event then
-- ...
```

The condition above will be true only on rising edge of the `CLK` signal, i.e. when the actual value of the signal is '1' and there was an event on it (the value changed recently).

Important Notes

- Sensitivity list or sensitivity set require an event on any of their signals, i.e. a change of the signal's value. A transaction (an assignment to a signal, no matter whether the same or a different value) is not enough.
- The concept of event relates to signals only. There is no "event on a variable" in VHDL.

Exit Statement

Formal Definition

The exit statement is used to finish or exit the execution of an enclosing loop statement. If the exit statement includes a condition, then the exit from the loop is conditional.

Complete definition: Language Reference Manual § 8.11.

Simplified Syntax

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

Description

The exit statement terminates entirely the execution of the loop in which it is located. The execution of the exit statement depends on a condition placed at the end of the statement, right after the `when` reserved word. When the condition is TRUE (or if there is no condition at all) the exit statement is executed and the control is passed to the first statement after the end loop (Example 1).

The loop label in the exit statement is not obligatory and can be used only in case of labeled loops. If no label is present then it is assumed that the exit statement relates to the innermost loop containing it. If an exit from a loop on a higher level of hierarchy is needed then the loop has to be assigned a label, which will be used explicitly in the exit statement. See Example 2.

Examples

Example 1

```
Loop_1: for count_value in 1 to 10 loop
  exit Loop_1 when reset= '1';
  A_1: A( count_value ) := '0';
end loop Loop_1;
A_2: B <= A;
```

At the beginning of each iteration of the `LOOP_1` loop, the `reset = '1'` condition is checked. If the condition is FALSE, then the rest of the loop is executed (in this case it is the assignment labeled A_1). Otherwise, the control is passed to the next statement after the loop, denoted with the A_2 label.

Example 2

```
Loop_X: loop
  a_v := 0;
  Loop_Y: loop
    Exit_1: exit Loop_X when condition_1;
    Output_1(a_v) := Input_1(a_v);
    a_v := a_v + 1;
    Exit_2: exit when condition_2;
  end loop Loop_Y;
  Assign_Y: B(i) <= Output_1(i);
  Exit_3: exit Loop_X when condition_3;
end loop Loop_X;
Assign_X: A <= B;
```

There are two nested loops in the above example. When the `condition_1` is TRUE, the `Exit_1` statement will be executed. This will cause termination of the `Loop_X` loop and moving the execution to `Assign_X`. Next, the `Loop_X` will be terminated because its label is explicitly listed in the exit statement.

If `condition_1` is not TRUE then the two assignments below it are performed and `condition_2` is checked. If it is TRUE, then the `Loop_Y` is exited. Since there is no loop label within the exit statement, therefore it relates to the innermost loop. As a result, the next statement to be executed will be `Assign_Y`. Finally, when the `condition_3` is TRUE the `Loop_X` is terminated. The `Exit_3` loop label could be skipped because this statement is within the boundaries of the exit `Loop_X` loop.

Important Notes

- The exit statement is often confused with the next statement. The difference between the two is that the exit statement "exits" the loop entirely, while the next statement skips to the "next" loop iteration (in other words, "exits" only the current iteration of the loop).

Expression

Formal Definition

A formula that defines the computation of a value.

Complete description: Language Reference Manual IEEE 1076-1993 § 7.1.

Syntax

```
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 ::= [ + | - ] 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 )
qualified_expression ::= type_mark'( expression )
  | type_mark'aggregate
```

Description

Expressions define the way in which values are computed. They consist of operands and operators. For example, the expression "1 + 2" adds two integer numbers. The numbers are operands and the plus sign is a pre-defined adding operator. Each operand has a value and a type of expression depends on the types of operands and the operation. The order in which the operations are performed depends on the priorities assumed in the language (see Operators for details).

If an expression is constructed using logical operators and, or, xor or xnor, it may contain a sequence of operations, but if operations nand or nor are used, an expression may contain only one operation of this type (Example 1). If more are needed, parentheses can be used.

The expression is evaluated from left to right preserving precedence of operations. If this precedence is to be changed, parentheses (introducing highest priority) can be applied (Example 2).

An expression may consist of one or more relations. A relation is created from operands in the form of a shift expression in which the operands are connected with each other by the relation operator. The shift expression is written by shifting simple expressions using the shift operators.

The simple expression consists of the sign operator with operands being the terms connected with each other by the adding operator.

The term consists of factors, which are connected with each other by the multiplication operation.

The factor may be so-called primary, possibly preceded by the abs or not operators or two primaries connected with an operator **.

The primary can be the name, literal, aggregate, function call, type conversion, allocator (see respective topics for details). Example 3 presents an example of a complex expression with several operators.

Examples

Example 1

```
variable A, B, C, D : BIT;
-- operator nand appears only once:
C := A nand B;
-- operators and, or can be used more than once in one expression:
A := '1' and B and C or D;
-- multilevel nand operation modeled with parentheses:
A := ( D nand B ) nand C;
```

Two logical operations (`nand` and `nor`) may appear only once in an expression, unless parentheses are used.

Example 2

```
A := '1' and ( B and ( C or D ) );
```

Without the parentheses, first the and logical operation of '1' and `B` would be performed, then `C` would be 'and-ed' to the result, and finally `D` would be 'or-ed' with the rest. With the parentheses, first `C` would be 'or-ed' with `D`, then 'and-ed' with `B` and finally logical and would be performed on the result of the operations and logical '1'.

Example 3

```
A1 := a * ( abs b ) + 10 <= 256;
```

Expression composed of several operands. `A1` must be of type BOOLEAN as the relation operator has higher precedence than arithmetic operations.

Important Notes

- Operators are defined for particular types of operands and this must be reflected in each expression.
- Different operators can be mixed in one expression as long as the operands are of correct (for each individual operator) type.

File Declaration

Formal Definition

A file declaration declares the file objects of a given file type.

Complete description: Language Reference Manual IEEE 1076-1993 §4.3.1.4.

Simplified Syntax

```
file identifier : subtype_indication [ file_open_information ];
file_open_information ::= [ open file_open_kind_expression ] is file_logical_name;
file_logical_name ::= string_expression
```

Description

The file declaration creates one or more file objects of the specified type. Such a declaration can be included in any declarative part in which the objects can be created, that is within architecture bodies, processes, blocks, packages or subprograms.

The optional parts of a declaration allow making an association between the file object and a physical file in the host file system. If these parts are attached to the declaration, the file is automatically opened for access. The optional file_open_kind_expression allows specifying how the physical file associated with the file object should be opened. The expression must have the predefined type file_open_kind value, which is declared in the standard package.

If the file_open_information is included in a given file declaration, then the file declared by the declaration is opened with an implicit call to FILE_OPEN when the file declaration is elaborated. If it is not, the file will not be opened.

If the file_open_kind_expression is not included in the file_open_information of a given file declaration, then the default value of READ_MODE is used during elaboration of the file declaration.

The file_logical_name must be an expression of predefined type STRING. The value of this expression is interpreted as a logical name for a file in the host system environment. The file_logical_name identifies an external file in the host file system that is associated with the file object. This association provides a mechanism for either importing data contained in an external file into the design during simulation or exporting data generated during simulation to an external file.

The files can also be declared in subprograms. In this case, the behavior is slightly different. The file is opened when the subprogram is called and is automatically closed again when the subprogram returns. Hence the file object, and its association with a physical file in the host file system, is purely local to the subprogram activation.

Examples

Example 1

```
type Integer_File is file of INTEGER;
file F1: Integer_File;
```

In this example no implicit FILE_OPEN is performed during elaboration.

Example 2

```
type Integer_File is file of INTEGER;
file F2: Integer_File is "test.dat";
```

The above example presents that an implicit call to FILE_OPEN is performed during elaboration. The OPEN_KIND parameter defaults to the READ_MODE mode.

Example 3

```
type Integer_File is file of INTEGER;
file F3: Integer_File open WRITE_MODE is "test.dat";
```

Example 3 presents an implicit call to FILE_OPEN being performed during elaboration. The OPEN_KIND parameter defaults to the WRITE_MODE mode.

- NOTE: All file objects associated with the same external file should be of the same base type.

File Type

Formal Definition

A type that provides access to objects containing a sequence of values of a given type. File types are typically used to access files in the host system environment. The value of a file object is the sequence of values contained in the host system file.

Complete description: Language Reference Manual IEEE 1076-1993 § 3.4.

Simplified Syntax

```
type type_name is file of type;
```

Description

The file type is used to define objects representing files in the host environment. The value of a file object is the sequence of values contained in the physical file.

The type mark in the file declaration defines the subtype of the values contained in the file. The subtype can be either constrained or unconstrained. The subtype cannot be based on a file type or an access type. If a composite type is used, the elements cannot be of an access type and in case of arrays, it must be a one-dimensional array. Example 1 shows several file type declarations.

When a file type is declared, several operations on objects of this type are implicitly defined. The list of the operations includes: opening a file (`FILE_OPEN`), closing a file (`FILE_CLOSE`), reading from a file (`READ`), writing to a file (`WRITE`) and checking the end of a file (`ENDFILE`).

Examples

Example 1

```
type POSITIVE_FILE is file of POSITIVE;
type BIT_VECTOR_FILE is file of BIT_VECTOR( 0 to 7 );
type STRING_FILE is file of STRING;
```

Example 2

```
type file_type is file of INTEGER;
file int_file : file_type open write_mode is "int_file.txt";
```

Example 3

```
type file_type is file of INTEGER;
file int_file: file_type;
file_open( int_file, "int_file.txt", write_mode );
```

Here, the first type declares a file of positive numbers, the second one - a file of 8-bit wide vectors of bits, and the third one - a file containing an indefinite number of strings of arbitrary length.

Important Notes

- File types are not supported by synthesis tools.

File Operations

Formal Definition

The operations for objects of a file type.

Complete description: IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1993 § 3.4.1.

Syntax

For a file type declared as

```
type FT is file of Some_Type;
```

the implicit operations are as follows:

```
procedure FILE_OPEN ( file anonymous: FT;
  External_Name: in STRING;
  Open_Kind: in FILE_OPEN_KIND := READ_MODE );
procedure FILE_OPEN ( Status: out FILE_OPEN_STATUS;
  file anonymous: FT;
  External_Name: in STRING;
  Open_Kind: in FILE_OPEN_KIND := READ_MODE );
procedure FILE_CLOSE ( file anonymous: FT );
procedure READ ( file anonymous: FT; Value: out Some_Type );
procedure WRITE ( file anonymous: FT; Value: in Some_Type );
function ENDFILE ( file anonymous: FT ) return BOOLEAN;
```

Description

FILE_OPEN Procedure

The FILE_OPEN procedures open an external file specified by the External_Name parameter and associate it with the file object F. If the call to FILE_OPEN is successful, the file object is said to be open and the file object has an access mode dependent on the value supplied to the Open_Kind parameter:

- If the value supplied to the Open_Kind parameter is READ_MODE, the access
- mode of the file object is read-only. In addition, the file object is initialized so that a subsequent READ will return the first value in the external file. Values are read from the file object in the order that they appear in the external file. If the value supplied to the Open_Kind parameter is WRITE_MODE, the
- access mode of the file object is write-only. In addition, the external file is made initially empty. Values written to the file object are placed in the external file in the order in which they are written. If the value supplied to the Open_Kind parameter is APPEND_MODE, the
- access mode of the file object is write-only. In addition, the file object is initialized so that values written to it will be added to the end of the external file in the order in which they are written.

In the second form of FILE_OPEN, the value returned through the Status parameter indicates the results of the procedure call:

- A value of OPEN_OK indicates that the call to FILE_OPEN was successful.

- If the call to FILE_OPEN specifies an external file that does not exist at the beginning of the call, and if the access mode of the file object passed to the call is write-only, then the external file is created. A value of STATUS_ERROR indicates that the file object already has an
- external file associated with it. A value of NAME_ERROR indicates that the external file does not exist (in the case of an attempt to read from the external file) or the external file cannot be created (in the case of an attempt to write or append to an external file that does not exist). This value is also returned if the external file cannot be associated with the file object for any reason. A value of MODE_ERROR indicates that the external file cannot be opened
- with the requested Open_Kind.

The first form of FILE_OPEN causes an error to occur if the second form of FILE_OPEN, when called under identical conditions, would return a Status value other than OPEN_OK.

A call to FILE_OPEN of the first form is successful if and only if the call does not cause an error to occur. Similarly, a call to FILE_OPEN of the second form is successful if and only if it returns a Status value of OPEN_OK.

FILE_CLOSE Procedure

If a file object F is associated with an external file, procedure FILE_CLOSE terminates access to the external file associated with F and closes the external file. If F is not associated with an external file, then FILE_CLOSE has no effect. In either case, the file object is no longer open after a call to FILE_CLOSE that associates the file object with the formal parameter F.

An implicit call to FILE_CLOSE exists in a subprogram body for every file object declared in the corresponding subprogram declarative part. Each such call associates a unique file object with the formal parameter F and is called whenever the corresponding subprogram completes its execution.

READ Procedure

Procedure READ retrieves the next value from a file; it is an error if the access mode of the file object is write-only or if the file object is not open.

For a file type declaration in which the type mark denotes an unconstrained array type, the same operations are implicitly declared, except that the READ operation is declared as follows:

```
procedure READ ( file F: FT; VALUE: out TM; LENGTH: out Natural );
```

The READ operation for such a type performs the same function as the READ operation for other types, but in addition it returns a value in parameter LENGTH that specifies the actual length of the array value read by the operation. If the object associated with formal parameter VALUE is shorter than this length, then only that portion of the array value read by the operation that can be contained in the object is returned by the READ operation, and the rest of the value is lost. If the object associated with formal parameter VALUE is longer than this length, then the entire value is returned and remaining elements of the object are unaffected by the READ operation.

An error will occur when a READ operation is performed on file F if ENDFILE(F) would return TRUE at that point.

WRITE Procedure

Procedure WRITE appends a value to a file; it is similarly an error if the access mode of the file object is read-only or if the file is not open.

ENDFILE Function

Function ENDFILE returns FALSE if a subsequent READ operation on an open file object whose access mode is read-only can retrieve another value from the file; otherwise, it returns TRUE. Function ENDFILE always returns TRUE for an open file object whose access mode is write-only. It is an error if ENDFILE is called on a file object that is not open.

NOTE: Predefined package TEXTIO is provided to support formatted human-readable I/O. It defines type TEXT (a file type representing files of variable-length text strings) and type LINE (an access type that designates such strings). READ and WRITE operations are provided in package TEXTIO that append or extract data from a single line. Additional operations are provided to read or write entire lines and to determine the status of the current line or of the file itself.

Examples

Example 1

```
type FILE_TYPE is file of INTEGER;
file INT_FILE: FILE_TYPE open WRITE_MODE is "int_file.txt";
```

Example 2

```
variable OPEN_STATUS : FILE_OPEN_STATUS;
type FILE_TYPE is file of INTEGER;
file FILE_READ : FILE_TYPE;
variable NUM: INTEGER;
file_open( OPEN_STATUS, FILE_READ, "int_file.txt", READ_MODE );
if OPEN_STATUS /= open_ok then
  report FILE_OPEN_STATUS'image( OPEN_STATUS )
    severity WARNING;
else
  read( FILE_READ, NUM );
end if;
```

Example 3

```
type BIT_VECTOR_FILE is file of BIT_VECTOR;
file VECTORS : BIT_VECTOR_FILE open APPEND_MODE is "vectors.dat";
variable NEXT_VECTOR : BIT_VECTOR( 7 downto 0 );
NEXT_VECTOR := "00101010";
write( VECTORS, NEXT_VECTOR );
file_close( VECTORS );
```

Example 4

```
type LOAD_FILE_TYPE is file of NATURAL;
variable STORAGE : NATURAL;
variable INDEX : INTEGER;
file LOAD_FILE: LOAD_FILE_TYPE open READ_MODE is "file.txt";
while not endfile( LOAD_FILE ) loop
  read( LOAD_FILE, STORAGE );
  INDEX := INDEX + 1;
end loop;
```

Floating Point Type

Formal Definition

Floating point type provides an approximation of the real number value.

Complete description: Language Reference Manual IEEE 1076-1993 § 3.1.4.

Simplified Syntax

```
type type_name is real_number_left_bound downto real_number_right_bound;
type type_name is real_number_left_bound to real_number_right_bound;
```

Description

A floating point type is a numeric type consisting of real numbers which values are constrained by a specified range.

There exists only one predefined floating point type: REAL. The range of the values for the type REAL are implementation-dependent, but it is required by the standard that it covers the values from -1.0E38 to +1.0E38.

A user-defined floating point type can be constructed on the basis of the predefined REAL type by constraining its range (Example 1). The bounds of the range of a user-defined floating point type should be in the form of locally static expression. The expression is classified as a locally static if it is possible to determine its value without running the code. The value of an expression used as a range for a floating point type must also be of floating point type, not necessarily the same for both bounds (Example 2). Negative bounds are allowed.

All floating point types (including user-defined) have the same set of arithmetic operators, namely: addition, subtraction, multiplication, division, absolute function and exponentiation.

Examples

Example 1

```
type Voltage_Level is range -5.5 to +5.5;
type Int_64K is range - 65536.00 to 65535.00;
```

Example 2

```
type APPROX_VALUES_DOWNTO is range ( 2.0 ** ( N + 1 ) ) - 1.0 downto 0.0;
type APPROX_VALUES_TO is range 0.0 to ( 2.0 ** ( N + 1 ) ) - 1.0;
```

Important Notes

- In order to add, subtract, multiply or divide integer object to/from a REAL object, type conversion of the integer object is needed. The only exception from this rule is multiplication and division of universal integer and universal real.
- The floating point types are not synthesizable by any of the existing tools. Their use is thus very limited.

Function

Complete Definition:

A function is a subprogram that returns a value.

Complete Description: Language Reference Manual section § 2.1, § 2.2, § 7.3.3.

Simplified Syntax

```
function function_name( parameters ) return type;
function function_name( parameters ) return type is
  [declarations]
begin
  sequential statements
end [function] [function_name];
```

Description

The function is a subprogram that either defines an algorithm for computing values or describes a behavior. The important feature of functions is that they are used as expressions that return values of specified type. This is the main difference from another type of subprograms: procedures, which are used as statements.

The result returned by a function can be of either scalar or complex type.

Functions can be either pure (which is default) or impure. Pure functions always return the same value for the same set of actual parameters. Impure functions may return different values for the same set of parameters. Additionally, an impure function may have "side effects", like updating objects outside of their scope, which is not allowed for pure functions.

The function definition consists of two parts:

- function declaration, which consists of name, parameter list and type of the values returned by the function;
- function body, which contains local declarations of nested subprograms, types, constants, variables, files, aliases, attributes and groups, as well as sequence of statements specifying the algorithm performed by the function.

The function declaration is optional and function body, which contains a copy of it, is sufficient for correct specification. However, if a function declaration exists, the function body declaration must appear in the given scope.

FUNCTION DECLARATION

The function declaration can be preceded by an optional reserved word `pure` or `impure`, denoting the character of the function. If the reserved word is omitted it is assumed by default that the function is pure.

The function name, which appears after the reserved word `function`, can be either an identifier or an operator symbol (if the function specifies the operator). Specification of new functions for existing operators is allowed in

VHDL and is called operator overloading. See respective topic for details.

The parameters of the function are by definition inputs and therefore they do not need to have the mode (direction) explicitly specified. Only constants, signals and files can be function parameters. The object class is specified by a reserved word (`constant`, `signal`, or `file`, respectively) preceding the parameter's name. If no reserved word is used, it is assumed by default that the parameter is a constant.

In case of signal parameters the attributes of the signal are passed into the function, except for 'STABLE, 'QUIET, 'TRANSACTION and 'DELAYED, which may not be accessed within the function.

If a file parameter is used, it is necessary to specify the type of the data appearing in the opened file.

Example 2 contains several examples of function declarations.

FUNCTION BODY

Function body contains a sequence of statements that specify the algorithm to be realized within the function. When the function is called, the sequence of statements is executed.

A function body consists of two parts: declarations and sequential statements. At the end of the function body, the reserved word `end` can be followed by an optional reserved word `function` and the function name. Example 2, Example 3, and Example 4 illustrate the function bodies.

IMPURE FUNCTIONS

If a function is explicitly specified as an impure (which is denoted with the reserved word `impure`, preceding the function declaration) it may return different results in different calls even with the same parameters. See Example 5.

Examples

Example 1

```
type Int_Data is file of NATURAL;
function Func_1( A, B, X : REAL )return REAL;
function "*" ( a,b : Integer_new ) return Integer_new;
function Add_Signals ( signal In1, In2 : REAL ) return REAL;
function End_Of_File ( file File_name : Int_Data ) return BOOLEAN;
```

The first function above is called `Func_1`, it has three parameters `A`, `B` and `X`, all of REAL type and returns a value also of REAL type.

The second function defines a new algorithm for executing multiplication. Note that the operator is enclosed in double quotes and plays the role of the function name.

The third function is based on signals as input parameters, which is denoted by the reserved word `signal` preceding the parameters.

The fourth function declaration is a part of the function checking for end of file, consisting of natural numbers. Note that the parameter list uses the Boolean type declaration.

Example 2

```
function Transcode_1( Value: in BIT_VECTOR( 0 to 7 ) ) return BIT_VECTOR is
begin
  case Value is
    when "00000000" => return "01010101";
    when "01010101" => return "00000000";
    when others => return "11111111";
  end case;
end function;
```

The case statement has been used to realize the function algorithm. The formal parameter appearing in the declaration part is the `Value` constant, that is a parameter of the BIT_VECTOR type. This function returns a value of the same type.

Example 3

```
function Func_3( constant A, B, X : REAL )return REAL is
begin
  return A * ( X ** 2 ) + B;
end function;
```

The formal parameters: A, B and X are constants of the REAL type. The value returned by this function is a result of calculating the `A * X**2 + B` expression and it is also of the REAL type.

Example 4

```
function Func_4( constant A, B, Step, Left_Bound, Right_Bound : in REAL )return REAL is
  variable counter, max, temp: REAL;
begin
  counter := Left_Bound;
  max := Func_3( A, B, counter );
  L1: while counter <= Right_Bound loop
    temp := Func_1( A, B, counter );
    if temp > max then
      max := temp;
    end if;
    counter := counter + Step;
  end loop L1;
  return max;
end function;
```

The fourth example is much more complicated. It calculates the maximum value of the `Func_1` function.

All the formal parameters are constants of the REAL type. When the function is called, the `A` and `B` values appearing in the function are passed, `Step` is a determinant of calculation correctness. The `Left_Bound` and `Right_Bound` values define the range in which we search for the maximum value of the function.

Inside the function body are contained definitions of variables `counter`, `max` and `temp`. They are used in the simple algorithm, which calculates all the function values in a given range and storing the maximum value returned by the function.

Example 5

```
variable number: INTEGER := 0;
impure function Func_5( A: INTEGER ) return INTEGER is
  variable counter: INTEGER;
begin
  counter := A * number;
  number := number + 1;
  return counter;
end function;
```

`Func_ 5` is an impure function; its formal parameter A and returned value are constants of the INTEGER type. When the function is invoked, output value depends on the variable number declared outside the function.

The number variable is additionally updated after each function call (it increases its value by 1). This variable affects the value calculated by the function, that is why the out function value is different for the same actual parameter value.

Important Notes

- Functions can be called recursively.
- Function body may not contain a wait statement or a signal assignment.
- Subprograms (functions and procedures) can be nested.

Generate Statement

Formal Definition

A mechanism for iterative or conditional elaboration of a portion of a description.

Complete Description: Language Reference Manual § 9.7.

Simplified Syntax

```
label: for parameter in range generate
  [{ declarations }
begin]
  { concurrent_statements }
end generate [label];

label: if condition generate
  [{ declarations }
begin]
  { concurrent_statements }
end generate [label];
```

Description

The generate statement simplifies description of regular design structures. Usually it is used to specify a group of identical components using just one component specification and repeating it using the generate mechanism.

A generate statement consists of three main parts:

- generation scheme (either for scheme or if scheme);
- declarative part (local declarations of subprograms, types, signals, constants, components, attributes, configurations, files and groups);
- concurrent statements.

The generation scheme specifies how the concurrent structure statement should be generated. There are two generation schemes available: for scheme and if scheme.

The for generation scheme is used to describe regular structures in the design. In such a case, the generation parameter and its scope of values are generated in similar way as in the sequential loop statement. Example 1 illustrates this concept with N-bit binary counter created by the generate statement that instantiates N D-type flip-flops (Figure 1).

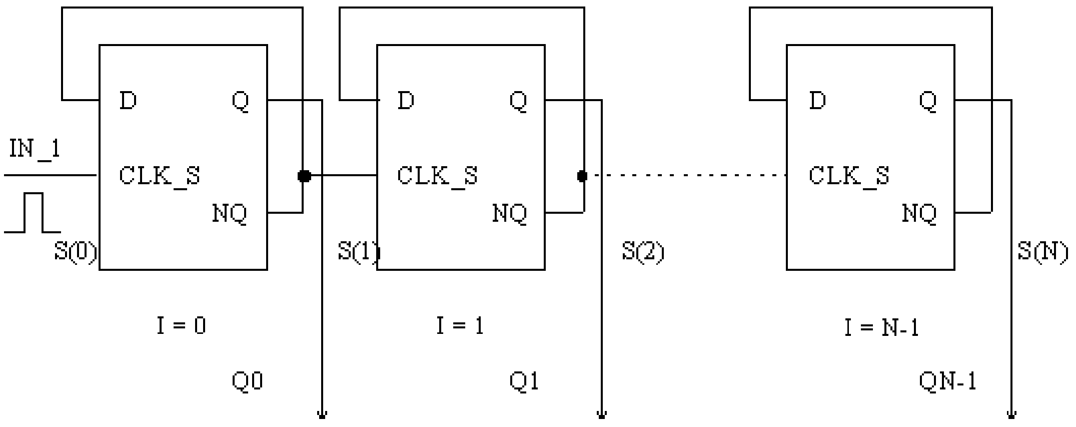


Fig. 1. The N-bit binary counter counting forward.

It is quite common that regular structures contain some irregularities. In such cases, the if scheme is very useful. Example 2 describes the synchronous decimal counter that consists of JK flip-flops and NAND gates (Fig. 2). In that structure there are some irregularities related to the connection of the next level of flip-flops.

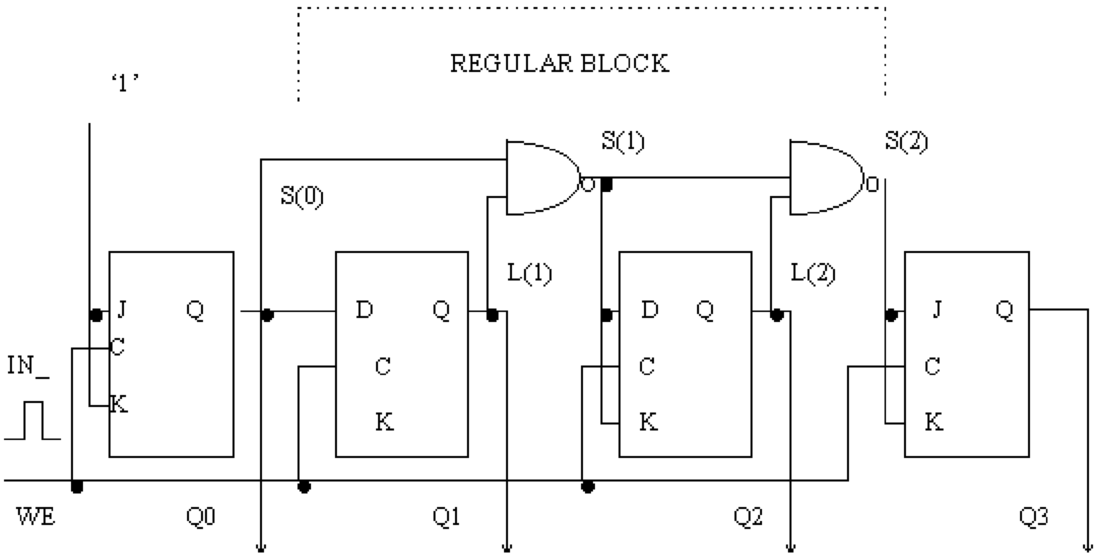


Fig. 2. The 8421 BCD counter counting forward.

A generate statement may contain any concurrent statement: process statement, block statement, concurrent assertion statement, concurrent procedure call statement, component instantiation statement, concurrent signal assignment statement, and another generate statement. The latter mechanism allows nesting the regular design structures and forming multidimensional arrays of components.

Examples

Example 1

```
entity D_FF is
  port(
    D, LK_S : in BIT;
    Q : out BIT := '0';
    NQ : out BIT := '1' );
end entity;

architecture A_RS_FF of D_FF is
begin
  BIN_P_RS_FF: process( CLK_S )
  begin
    if rising_edge(CLK_S) then
      Q <= D;
      NQ <= not D;
    end if;
  end process;
end architecture;

entity COUNTER_BIN_N is
  generic(
    N : INTEGER := 4 );
  port(
    Q : out BIT_VECTOR( 0 to N-1 );
    IN_1 : in BIT );
end entity;

architecture behavioral of COUNTER_BIN_N is
  component D_FF
    port(
      D, CLK_S : in BIT;
      Q, NQ : out BIT );
  end component D_FF;
  signal S : BIT_VECTOR( 0 to N );
begin
    S(0) <= IN_1;
    G_1: for I in 0 to N-1 generate
      D_Flip_Flop: D_FF port map(
        S( I+1 ), S(I), Q(I), S( I+1 ) );
    end generate;
end architecture;
```

First, a specification of a D flip-flop is given which will be used by the component declaration. The generate statement here is used to define a counter of arbitrary length, determined only by its generic parameter (set here to 4).

Example 2

```
-- the 8421 BCD counter
entity COUNTER_BCD is
  port(
    IN_1 : in BIT;
    Q : out BIT_VECTOR( 0 to 3 ) );
end entity;

architecture STRUCT of COUNTER_BCD is
  component D_FF
    port(
      J, K, CLK_S : in BIT;
      Q : out BIT );
  end component;
  component NAND_GATE
    port(
      IN1, IN2 : in BIT;
      OUT1 : out BIT );
  end component;
  signal S: BIT_VECTOR( 0 to 2 );
  signal L: BIT_VECTOR( 0 to 1 );
begin
  D_FF_0: D_FF port map( '1', 1', N_1, S(0) );
  Gen_1: for I in 1 to 3 generate
    Gen_2: if I = 1 or I = 2 generate
      D_FF_I: D_FF port map( S( I-1 ), ( I-1 ), IN_1, L( I-1 ) );
      NAND_I: NAND_GATE port map( S( I-1 ), ( I-1 ), S(I) );
      Q(I) <= L( I-1 );
    end generate;
    Gen_3: if I = 3 generate
      D_FF_3: D_FF port map( S( I-1 ), ( I-1 ), IN_1, Q(I) );
    end generate;
  end generate;
  Q(0) <= S(0);
end architecture;
```

Nested generate statements have been used here in order to shorten the description. The outermost generate statement specifies the complete counter, which component parts are generated by the inner statements depending on the index of the component (flip-flop) inside the counter.

Important Notes

- Each generate statement must have a label.
- If the generate statement does not contain any local declarations then the reserved word `begin` need not to be used.
- Generate statements can be nested.

Generic

Formal Definition

An interface constant declared in the block header of a block statement, a component declaration, or an entity declaration. Generics provide a channel for static information to be communicated to a block from its environment. Unlike constants, however, the value of a generic can be supplied externally, either in a component instantiation statement or in a configuration specification.

Complete description: Language Reference Manual IEEE 1076-1993 § 1.1.1.1.

Simplified Syntax

```
generic( generic_interface_list );
```

Description

Generics support static information to blocks in a similar way as constants, but unlike the constants the values of generics can be supplied externally. Similar to ports, they can be declared in entities and component declarations, and always before ports.

Values supported by generics declared in an entity can be read either in entity or in architecture associated with the entity. In particular, a generic can be used to specify the size of ports (Example 1), the number of subcomponents within a block, the timing characteristics of a block (Example 2), physical characteristics of a design, width of vectors inside an architecture, number of loop iterations, etc. In general, generic can be treated inside an architecture in the same way as constant.

Examples

Example 1

```
entity CPU is
  generic(
    BusWidth : INTEGER := 16 );
  port(
    Data_Bus : inout STD_LOGIC_VECTOR( BusWidth-1 downto 0 ) );
  -- ...
end entity;
```

The generic value `BusWidth` is used here to declare the width of the port `Data_Bus`, and can be successively in all declarations of buses inside associated architecture(s). This way the user supplies only one value, which parameterizes complete design.

Example 2

```
entity Gen_Gates is
  generic(
    Delay : TIME := 10 ns );
  port(
    In1, In2 : in STD_LOGIC;
    Output : out STD_LOGIC );
end entity;

architecture Gates of Gen_Gates is
begin
  -- ...
  Output <= In1 or In2 after Delay;
  -- ...
end architecture;
```

The `Delay` generic value specifies here the delay through a device or part of the device.

Important Notes

- In most synthesis tools only generics of type integer are supported.

Group

Formal Definition

A named collection of named entities. Groups relate different named entities for the purposes not specified by the language. In particular, groups may be decorated with attributes.

Complete description: Language Reference Manual IEEE 1076-1993 § 4.6, § 4.7.

Simplified Syntax

```
group group_template_name is ( entity_class_list );
group group_name : group_template_name ( group_constituent_list );
```

Description

The user-defined attributes are connected individually with each named entity. That is why each separate named entity can have its own attributes. In case when the user wants to assign information to several related named entities, he /she should define a group consisting of these units, and then specify attributes for the entire group. The set of units with the specified characteristics can be defined by means of group declaration. The group declaration in turn requires group template declaration to be defined earlier.

The Group Template Declaration

The group template declaration defines pattern of a group connecting named entities with the specified class. The set of possible entity classes contains entity, architecture, configuration, procedure, function, package, type, subtype, constant, signal, variable, component, label, literal, units, group, and file. Each entity class entry defines an entity class that may appear at that particular position in the group type (Example 1).

The box symbol (<>) can be used together with the name of an entity class to allow zero or more units belonging to this group (Example 2). If such a declaration appears on the list, it must be the last one.

The Group Declaration

Group declaration connects named entities having the specified characteristics.

The group declaration consists of identifier, group template name and group constituent list. The identifier represents a group, the group template name indicates group's template declaration, and the group constituent indicates the chosen named entities belonging to this group (Example 3).

The Attributes of Groups

The attribute specification of a group is realized in a similar way to other attribute specifications. First, the attribute of a given type is declared and it is specified for the given group in its declaration part which contains declaration of the particular group (Example 4).

Examples

Example 1

```
group Variable_group is ( variable, variable );
```

The `Variable_group` template group declaration creates two variables which serve as a pattern for the variables belonging to this group.

Example 2

```
group Component_group is ( component <> );
```

The `Component_group` pattern declaration creates a group, which consists of a component list of arbitrary length.

Example 3

```
group Input_pair : Variable_group( A1, A2 );
```

The `Input_pair` group declaration creates a group, which consists of A1 and A2 variables and is based on the Variable_group group template (declared in Example 1).

Example 4

```
function Compute_Values( A, B: INTEGER ) return BOOLEAN is
  variable A1, A2: INTEGER;
  group Variable_group is ( variable, variable );
  group Input_pair : Variable_group ( A1, A2 );
  attribute Input_name : STRING;
  attribute Input_name of Input_pair : group is "Input variables";
begin
  -- ...
end function;
```

The value of the attribute `Input_name` for the group `Input_pair` is equal to "Input variables".

Important Notes

- Groups can be nested.

Guard

Formal Definition

A Boolean-valued expression associated with a block statement that controls assignments to guarded signals within a block. A guard expression defines an implicit signal GUARD that may be used to control the operation of certain statements within the block.

Complete description: Language Reference Manual IEEE 1076-1993 § 4.3.1.2, § 9.1, § 9.5.

Simplified Syntax

```
some_signal_in_a_block <= guarded expression;
```

Description

The characteristic feature of the block statement is the guard expression. It is a logical expression of the `BOOLEAN` type, declared implicitly after the reserved word `block` whenever a guarded expression appears inside the block (Example 1).

The guard expression implies a signal named 'guard' at the beginning of the block declaration part. This signal can be read as any other signal inside the block statement but no assignment statement can update it. This signal is visible only within the given block. Whenever a transaction occurs on any of the signals on the right hand side of the guard expression, the expression is evaluated and the 'guard' signal is immediately updated. The 'guard' signal takes on the TRUE value when the value of the guard expression is true. Otherwise, 'guard' takes on the FALSE value.

The 'guard' signal may also be declared explicitly as a Boolean signal in the block statement. The advantage of this approach is that more complex (than a simple Boolean expression) algorithm to control the guard signal can be used. In particular, a separate process (Example 2) can drive the guard signal.

If there is no guard expression and the guard signal is not declared explicitly, then by default the guard signal is always true.

The guard signal is used to control so called guarded concurrent signal assignment statements contained inside the block. Each such statement contains the reserved word `guard` placed after the symbol "<=". They assign a new value to the signal only when the guard signal is true. Otherwise, the signal assignment statement does not change the value of the given signal. In Example 1, the signal OUT_1 will take on the value of not IN_1 only when the value of the expression CLK'EVENT and CLK='1' will be true.

Examples

Example 1

```
RISING_EDGE : block ( rising_edge( CLK ) )
begin
  OUT_1 <= guarded not IN_1;
  -- ...
end block;
```

The assignment to the signal `OUT_1` is guarded, which introduces the implicit `GUARD` signal into the block.

Example 2

```
ALU : block
  signal GUARD : BOOLEAN := FALSE;
begin
  OUT_1 <= guarded not IN_1;
  -- ...
  P_1: process
  begin
    GUARD <= TRUE;
    -- ...
  end process;
end block;
```

Signal `GUARD` is declared explicitly and can be assigned a value like any other signal.

Important Notes

- Guarded blocks are usually not supported for synthesis.

Identifier

Formal Definition

Names that identify various VHDL named entities.

Complete description: Language Reference Manual IEEE 1076-1993 § 13.3.

Syntax

```
identifier ::= basic_identifier | extended_identifier
basic_identifier ::= letter { [ underline } letter_or_digit }
letter_or_digit ::= letter | digit
letter ::= upper_case_letter | lower_case_letter
extended_identifier ::= \graphic_character { graphic_character } \
```

Description

Identifiers are used both as names for VHDL objects, procedures, functions, processes, design entities, etc., and as reserved words. There are two classes of identifiers: basic identifiers and extended identifiers.

The basic identifiers are used for naming all named entities in VHDL. They can be of any length, provided that the whole identifier is written in one line of code. Reserved words cannot be used as basic identifiers (see Reserved Word section for the complete list of VHDL reserved words). Underscores are significant characters in an identifier and basic identifiers may contain underscores, but it is not allowed to place an underscore as a first or last character of an identifier. Moreover, two underscores side by side are not allowed as well. Underscores are significant characters in an identifier.

The extended identifiers were included in VHDL '93 in order to make the code more compatible with tools which make use of extended identifiers. The extended identifiers are braced between two backslash characters. They may contain any graphic character (including spaces and non-ASCII characters), as well as reserved words. If a backslash is to be used as one of the graphic characters of an extended literal, it must be doubled. Upper- and lower-case letters are distinguished in extended literals.

Examples

Example 1

```
-- legal identifiers
Decoder_1 FFT Sig_N Not_Ack
\signal\ \C:\\Cads\ \Signal @#\
```

All above identifiers are legal in VHDL '93. Note that a single backslash inside an extended name is denoted by two backslashes.

Example 2

```
-- illegal identifiers
_Decoder_1 2FFT Sig_#N Not-Ack
```

No VHDL tool would accept any of the four identifiers above. The errors are as follows: underscore at the beginning of the first identifier, digit at the beginning of the second, special character (#) inside the third and hyphen (special character as well) in the fourth.

Important Notes

- A basic identifier must begin with a letter.
- No spaces are allowed in basic identifiers.
- Basic identifiers are not case sensitive, i.e. upper- and lower-case letters are considered identical.
- Basic identifiers consist of Latin letters (a..z), underscores (_) and digits (0..9). It is not allowed to use any special characters here, including non-Latin (language-specific) letters.

If Statement

Definition:

The if statement is a statement that depending on the value of one or more corresponding conditions, selects for execution one or none of the enclosed sequences of statements,

Complete definition: Language Reference Manual § 8.7.

Simplified Syntax

```
[ label: ] if condition then
  sequential_statements
{ elsif condition then
  sequential_statements }
[ else
  sequential_statements ]
end if [ label ];
```

Description

The if statement controls conditional execution of other sequential statements. It contains at least one Boolean condition (specified after the `if` keyword). The remaining conditions are specified with the `elsif` clause. The `else` clause is treated as `elsif TRUE then`. Conditions are evaluated one by one until any of them turns to be true or there are no more conditions to be checked for. When a condition is true then the sequence of statements specified after the `then` clause is executed. If no condition is met then the control is passed to the next statement after the if statement.

The if statement can be used in a simplified form, often called if-then statement, where neither `elsif` nor `else` clause is specified (Example 1).

In case when meeting a condition should cause some statements to be executed, but when it is not met some other actions should be performed, the else clause is used (Example 2).

The elsif clause is used when different nested conditions have to be checked (Example 3). This can be used for prioritizing conditions and signals.

The if statements can be nested (Example 4).

Examples

Example 1

```
I1: if Status_Signal = hold then
  A1: Outputs <= 'X';
end if I1;
```

The assignment will be realized only when the condition `Status_Signal = hold` is true. Otherwise, the statement that follows the `end if I1` will be executed.

Example 2

```
function AND_FUNC( x, y: in BIT ) return BIT is
begin
  I2: if x= '1' and y= '1' then
    return '1';
  else
    return '0';
  end if;
end function;
```

When the variables x and y are both equal to '1', then the function returns the value '1'. Otherwise, '0' is returned.

Example 3

```
signal Code_of_Operation : BIT_VECTOR( 1 downto 0 );
I3: if Code_of_Operation(1)= '1' then
  F := Operand_1 + Operand_2;
elsif Code_of_Operation(0)= '1' then
  F := Operand_1 - Operand_2;
else
  F := ( others => '0' );
end if I3;
```

In this example, the bit number 1 of the `Code_of_Operation` has a higher priority than bit number 0. When the bit number 1 has a '1'value, then the two operands are added. If not (i.e. it is '0'), then the bit number 0 is checked. If it is '1', then the two operands are subtracted. Otherwise, when both bits of the `Code_of_Operation` are '0', the F signal is assigned all zeros.

Example 4

```
I4: if Status = RUN then
  I5: if Code_of_Operation = CONCATENATE then
    F := Operand_1 & Operand_2;
  else
    F := ( others => '0' );
  end if I5;
  Output_1 <= F;
end if I4;
```

Nesting of if statements is legal, but you should be careful not to confuse the statement levels.

Important Notes

- One of the most typical errors with the if statements is to skip the space between end and if, at the end of the statement, and writing it as endif.
- if is a sequential statement that cannot be used in the concurrent statements section of an architecture. If assigning a new value to a signal must be influenced by a specific condition, then a conditional signal

assignment should be used (which in turn cannot be used inside processes and procedures).

Integer Type

Definition:

The integer type is a scalar whose set of values includes integer numbers of the specified range.

Complete description: Language Reference Manual IEEE 1076-1993 § 3.1.2.

Simplified Syntax

```
type type_name is integer_left_bound to integer_right_bound;
type type_name is integer_left_bound downto integer_right_bound;
```

Description

An integer type is a numeric type which consists of integer numbers within the specified range. There is a predefined INTEGER type, which range depends on the implementation, however, must cover at least the range -2147483648 to +2147483647.

A user-defined integer type can be constructed on the basis of the predefined INTEGER type by constraining its range (Example 1). The bounds of the range of a user-defined integer type should be in the form of a locally static expression. (In VHDL an expression is said to be locally static if it can be evaluated at compile time.) The value of an expression used as a range for an integer type must also be of integer type, not necessarily the same for both bounds (Example 2). Negative bounds are allowed.

All integer types (including user-defined) have the same set of arithmetic operators, defined in the package STANDARD, namely: addition, subtraction, multiplication, division, modulus, and remainder. In all cases both operands and the result are of the integer type.

Besides arithmetic operations, relations can also be checked on such integer operands as: equal, unequal, greater than, less than, greater or equal than, and less or equal than. In all cases, the result is Boolean.

Examples

Example 1

```
type Voltage_Level is range 0 to 5;
type Int_64K is range -65536 to 65535;
type WORD is range 31 downto 0;
```

An integer type can be defined either as an ascending or descending range.

Example 2

```
type MUX_ADDRESS is range ( 2 ** ( N + 1 ) ) - 1 downto 0;
```

The parameter `N` must have an explicitly stated value (e.g. as a constant).

Important Notes

- It is an error to assign to an integer object a value which is from outside its range.

Library Clause

Formal Definition

A library clause defines logical names for design libraries in the host environment.

Complete description: Language Reference Manual IEEE 1076-1993 § 11.2.

Simplified Syntax

```
library library_name;
```

Description

The library clause defines the logical names of design libraries, which are used by the design units. A library is a storage facility for previously analyzed design units. In practice, this relates mostly to packages.

When a package is to be used in a design, it has to be made visible to the design. In order to specify it, the library clause (making the library visible) and use clause (making particular declarations visible) must be used. See Use Clause for more details.

There are two predefined libraries, which are used implicitly in every design: STD and WORK. The first of them contains standard packages STANDARD and TEXTIO. The other is a working library, where all user-created and analyzed design units are stored.

User-specified packages are stored in the working library WORK.

Examples

```
library IEEE;
use IEEE.std_logic_1164.all;
```

Such declarations as in the above example must precede each design entity, which uses any of the declarations in the package std_logic_1164.

Important Notes

- A library specified in a library clause of the primary design unit (entity, configuration or package) is visible in each secondary unit (architecture or package body) associated to it.
- Library STD (containing packages STANDARD and TEXTIO) need not to be specified. Both packages are automatically included in every design unit.
- Package std_logic_1164 is specified in the library IEEE. The library IEEE clause must be used in design units, which will use this package.
- Library clause may contain more than one library names separated by commas.

Literal

Formal Definition

A value that is directly specified in a description of a design.

Complete description: Language Reference Manual IEEE 1076-1993 § 7.3.1.

Full Syntax:

```
literal ::= numeric_literal
  | enumeration_literal
  | string_literal
  | bit_string_literal
  | null
numeric_literal ::= abstract_literal | physical_literal
abstract_literal ::= decimal_literal | based_literal
decimal_literal ::= integer [ . integer ] [ exponent ]
integer ::= digit { [ underline ] digit }
exponent ::= E [ + ] integer | E - integer
based_literal ::= base # based_integer [ . based_integer ] # [ exponent ]
base ::= integer
based_integer ::= extended_digit { [underline] extended_digit }
extended_digit ::= digit | letter
physical_literal ::= [ abstract_literal ] unit_name
enumeration_literal ::= identifier | character_literal
character_literal ::= ' graphic_character '
string_literal ::= " { graphic_character } "
bit_string_literal ::= base_specifier " bit_value "
base_specifier ::= B | O | X
bit_value ::= extended_digit { [ underline ] extended_digit }
```

Description

There are five classes of literals: numeric literals, enumeration literals, string literals, bit string literals and the literal null.

Numeric literals

The class of numeric literals includes abstract literals (which include integer literals and real literals) and physical literals.

The difference between integer and real literals lies in the decimal point: a real literal includes a decimal point, while an integer literal does not. When a real or integer literal is written in the conventional decimal notation it is called a decimal literal. Example 2 presents several abstract literals - both integer and real.

If a number is written in the exponential form, the letter E of the exponent can be written either in lowercase or in uppercase. If the exponential form is used for an integer number, then zero exponent is allowed.

Abstract literals can be written in the form of based literals as well. In such a case the base is specified explicitly (in decimal literals the base is implicitly ten). The base in based literal must be at least two and at most sixteen. The base is specified in decimal notation.

The digits used in based literals can be any decimal digits (0..9) or a letter (either in upper or lower case). The meaning of based notation is as in decimal literals, with the exception of base. Several based literals are presented in Example 1.

A physical literal consists of an abstract numeric literal followed by an identifier that denotes the unit of the given physical quantity (Example 1).

Enumeration literals

The enumeration literals are literals of enumeration types, used in type declaration and in expressions evaluating to a value of an enumeration type. They include identifiers and character literals. Reserved words may not be used in identifiers, unless they are a part of extended identifiers which start and end with a backslash. See identifiers for more information on identifiers. Example 2 presents several enumeration literals.

String Literals

String literals are composed as a sequence of graphic characters (letters, digits, special characters) enclosed between two quotation marks (double quotes). They are usually used for warnings or reports which are displayed during simulation (Example 3).

Bit string literals

Bit string literals represent values of string literals that denote sequences of extended digits, range of which depends on the specified base.

The base specifier determines the base of the digits: letter B used as a base specifier denotes binary digits (0 or 1), letter O - octal digits (0 to 7) and letter X - hexadecimal (digits 0 to 9 and letters A to F, case insensitive). Underlines can be used to increase readability and have no impact on the value.

All values specified as bit string literals are converted into binary representation without underlines. Binary strings remain unchanged (only underlines are removed), each octal digit is converted into three bits and each hexadecimal into four bits (Example 4).

Examples

Example 1

```
-- Decimal literals:
14
7755
156E7
188.993
88_670_551.453_909
44.99E-22
-- Based literals:
16#FE# -- 254
2#1111_1110# -- 254
```

```
8#376# -- 254
16#D#E1 -- 208
16#F.01#E+2 -- 3841.00
2#10.1111_0001#E8 -- 1506.00
-- Physical literals:
60 sec
100 m
kohm
177 A
```

Abstract literals written in different forms: as decimal, based and physical literals.

Example 2

```
State0 Idle
TEST \test\
\out\ \OUT\
```

This example presents several enumeration literals. Note that such literals are not specified in double quotes. `out` is a reserved word and as such cannot be used as an identifier (enumeration literal). It may be, however, a part of an extended literal. Extended literals are case sensitive; therefore, \out\ and \ OUT\ are different literals.

Example 3

```
"ERROR"
"Both S and Q equal to 1"
"X"
"BB$CC"
"{LOW}"
"Quotation: ""REPORT..."""
```

A string literal may contain any number of characters (including special characters), as long as it fits on one line. It can also be a null string or a one-character long string. If a quotation sign has to be used in a string literal, it is denoted by two double quotes side by side.

Example 4

```
B"1111_1111" -- binary representation of a vector
B"11111111" -- binary representation of the same vector
X"FF" -- hexadecimal representation of the same vector
O"377" -- octal representation of the same vector
```

Four methods to denote decimal 255 with bit string literals. In all cases the literals will be converted into binary version, i.e. "11111111".

Important Notes

- Underline character can be used in decimal literals (inserted between adjacent digits) to improve readability. In such a case it has no other meaning than just a space.
- Abstract literals cannot contain any spaces, not even between constituents of the exponent.

- A string literal must fit on one line. If longer sequence of characters in required, it may be obtained by concatenating shorter literals.

Loop Statement

Formal Definition

Statement that includes a sequence of statements that is to be executed repeatedly, zero or more times.

Complete description: Language Reference Manual IEEE 1076-1993 section § 8.9.

Simplified Syntax

```
[ loop_label: ] while condition loop
  sequence_of_statements
end loop [ loop_label ];

[ loop_label: ] for loop_parameter in range loop
  sequence_of_statements
end loop [ loop_label ];
```

Description

The loop statement contains a sequence of statements, which are supposed to be repeated many times. The statement also lists the conditions for repeating the sequence or specifies the number of iterations.

A loop statement can have several different forms depending on the iteration scheme preceding the reserved word `loop`. In its simplest form, no iteration scheme is specified and the loop is repeated indefinitely (Example 1).

In order to exit from an infinite loop, an exit statement has to be used. See exit statement for details. An exit statement can be specified with a condition that must be met to exit the loop (Example 2).

Instead of specifying an infinite loop with a conditional exit statement, a while loop can be used. In such a case the reserved word `while` with a condition precede the keyword `loop`. The sequence of statements inside the loop will be executed if the condition of the iteration scheme is true. The condition is evaluated before each execution of the sequence of statements. When the condition is false, the loop is not entered and the control is passed to the next statement after the end loop clause (Example 3).

Another iteration scheme is useful when a discrete range can define the number of iterations. In this case the keyword `for` with a loop parameter precede the keyword `loop`. The header of the loop also specifies the discrete range for the loop parameter. In each iteration the parameter takes one value from the specified range, starting from the leftmost value within the range (Example 4).

The discrete range can also be declared in the form of a discrete type (Example 5), including an enumerated type.

Examples

Example 1

```
signal Clock : BIT := '0';
-- ...
Clk_1: process
begin
  L1: loop
    Clock <= not Clock after 5 ns;
  end loop L1;
end process;
```

The process (which is a clocking signal generator) contains a loop without an iteration scheme, which will iterate indefinitely.

Example 2

```
L2: loop
  A := A + 1;
  exit L2 when A > 10;
end loop L2;
```

The infinite loop becomes in practice a finite, as the iterations will terminate as soon as the variable A becomes greater than 10.

Example 3

```
Shift_3: process ( Input_X )
  variable i : POSITIVE := 1;
begin
  L3: while i <= 8 loop
    Output_X(i) <= Input_X( i + 8 );
    i := i + 1;
  end loop;
end process;
```

The loop `L3` will be repeated as long as the value of the variable `i` is not greater than 8. When `i` reaches the value of 9 the loop is no longer repeated.

Example 4

```
Shift_4: process ( Input_X )
begin
  L4: for count_value in 1 to 8 loop
    Output_X( count_value ) <= Input_X( count_value + 8 );
  end loop;
end process;
```

In the above example the loop statement parameter `count_value` will cause the loop to execute 8 times, with the value of `count_value` changing from 1 to 8.

Example 5

```
Shift_5: process ( Input_X )
  subtype Range_Type is POSITIVE range 1 to 8;
begin
  L5: for count_value in Range_Type loop
    Output_X( count_value ) <= Input_X( count_value +8 );
  end loop L5;
end process;
```

The subtype `Range_Type` defines the range of integer values from 1 to 8. The parameter `count_value` changes according to this specification, i.e. from 1 to 8.

Important Notes

- The parameter for a 'for' loop does not need to be specified - the loop declaration implicitly declares it.
- The loop parameter is a constant within a loop, which means that it may not be assigned any values inside the loop.

Name

Formal Definition

A property of an identifier with respect to some named entity. Each form of declaration associates an identifier with a named entity. In certain places within the scope of a declaration, it is valid to use the identifier to refer to the associated named entity; these places are defined by the visibility rules. At such places, the identifier is said to be the name of the named entity.

Complete description: Language Reference Manual IEEE 1076-1993 § 4, § 6.1.

Simplified Syntax

```
simple_name ::= identifier
selected_name ::= prefix . suffix
prefix ::= name | function_call
suffix ::= simple_name
  | character_literal
  | operator_symbol
  | all
indexed_name ::= prefix ( expression {, expression } )
slice_name ::= prefix ( discrete_range )
attribute_name ::= prefix [ signature ] ' attribute_designator [ ( expression )]
```

Description

Any declaration that introduces a named entity defines an identifier which enables reference to such an entity by using this identifier. However, it is not always possible to refer to an entity or part of it by using just its identifier. More general form of reference to entities is by a name. Names can also indicate objects of the access type, elements of the composite type, parts of the composite object or unit attributes which have an identifier in their declaration. The name can have any of the following forms:

- simple name
- operator symbol
- selected name
- indexed name
- slice name
- attribute name

The selected name, indexed name, slice name and attribute name contain prefix, which can be also a name or a function call. Prefix of selected name, indexed name or slice name must indicate a composite unit, which is composed of other units (Example 1).

The prefix of an attribute name can indicate any unit for which such an attribute exists (Example 2).

SIMPLE NAME AND OPERATOR SYMBOL

The simple name consists only of the identifier, which is assigned to a given unit in its declaration. The simple name can also indicate an alias of a given unit (Example 3).

The operator symbol is a string literal, which indicates the function declared for such an operator symbol (Example 4).

THE SELECTED NAME

The selected name serves for indicating a unit declared inside another unit or in a design library. A selected name consists of the prefix, which is the name of a composite unit, followed by a dot (.) and then by a suffix which is either the name of the element (if referenced individually) or the reserved word `all` (if all elements in the composite unit are referenced to).

The selected name can indicate all the units declared in a library and in a package. It can also define elements of a record, as well as objects indicated by pointers. The prefix is the composite unit in which the units are declared. The suffix can be a simple name, character literal, the operator symbol or the reserved word `all`. In the latter case, the name refers to all elements declared in a unit. Several selected names are presented in Example 5.

The prefix may also have a form of a selected name (with two dots in total), making the reference more complex. This form is used for example in the use clause. See Example 6.

The prefix in the selected name can indicate design entity, architecture body, subprogram, block, process, generate statement or loop statement. The entity indicated by the suffix of this name must be declared inside a given construct (Example 7).

THE INDEXED NAME

The indexed name indicates an element of an array, which is indicated by an expression list forming the array index. The number of expressions in the list must match the array dimension and the expressions values must fit within the respective index ranges (Example 8).

The index name can have the same form as the function call. In such a case, the name's interpretation depends on the context. If the interpretation of such a name is impossible, then the name is ambiguous.

THE SLICE NAME

The slice name allows indicating a part of a one-dimensional array. The part of the object indicated by the slice name is an object of the same class, e.g. slice name of variable is also a variable. The slice name consists of two elements: a prefix and a discrete range. The prefix is the name of the array, while discrete range defines which elements of the object are accessed through the slice name (Example 9).

The discrete range must be compatible with the array range, i.e. the range bounds must not be outside the range declared for the object and the direction must be the same as in the object's declaration.

The slice name is called a null slice if its discrete range is a null range. Null slices are illegal.

THE ATTRIBUTE NAME

The attribute name consists of two elements: a prefix and an attribute designator (name). The prefix indicates a unit, for which the attribute is specified and the attribute designator specifies the name of the attribute (Example

10).

The specification of an attribute name may contain an expression, but this is restricted only to those predefined attributes which require such expression (see Attributes (predefined) for details). User-defined attributes may not be used with expressions.

The prefix can also indicate the unit alias. In such a case, attribute name defines the attribute of the unit and not the unit's alias. There are only three exceptions from this rule: attributes SIMPLE_NAME, PATH_NAME and INSTANCE_NAME (Example 11).

If the prefix indicates a subprogram, enumeration literal or an alias of any of the two, then the attribute name can also contain the signature, which helps to resolve any possible ambiguities arising from multiple objects with the same name (Example 12).

Examples

Example 1

```
C.X
A( 1, 0 )
B( 1 to 2 )
```

`C` is an object of a record type, `A` is an object of an array type or a function call (the two numbers can be either indices of a two-dimensional array or two parameters of a function call) and `B` is an object of an array type.

Example 2

```
D'STABLE( 5 ns )
```

Prefix `D` indicates a signal of any type.

Example 3

```
variable E : BIT_VECTOR( 0 to 3 );
alias F : BIT_VECTOR( 0 to 3 ) is E;
E := "0000";
F := "1111";
```

`E` is a simple name which indicates the `E` variable, and `F` is a simple name which indicates alias of the `E` variable.

Example 4

```
function "+"( a, b : INTEGER ) return REAL;
G := "+"( 7, 7 );
```

In this example, `"+"` is an overloaded operator defined by the associated function (declaration of which is given above the call to the operator).

Example 5

```
WORK.N_GATES
DATA_RECORD.DAY
Package_Operators."*"
STD_ULOGIC.'X'
NMOS_GATES.all
REC_PTR.all
```

The first name gives an access to the package `N_Gates` in the `Work` library. The second refers to the element `Day` of the record `Data_Record`. Note that both names are constructed in the same way and their correct interpretation is possible only from the context (not presented here). The third name relates to operation of a package. Fourth is a value of a type. Finally, the last two names relate to all declarations in a package and all objects addressed by the value of an access type, respectively.

Example 6

```
A_Library.Package_Operators."+"
STD_LOGIC.STD_ULOGIC.'U'
```

The first complex name gives an access to the `"+"` operator defined in the `Package_Operators` package stored in the `A_Library` library. The second example refers to the value `"U"` declared by the STD_ULOGIC type within STD_LOGIC package.

Example 7

```
B_1: block
  signal Sig_X: BIT := '0';
begin
  -- ...
  Gen_1: for i in 1 to 10 generate
    signal Sig_X : BIT;
  begin
    -- ...
    Sig_X <= B_1.Sig_X;
  end generate;
end block;
```

The access to the `Sig_X signal` declared in the `B_1` block statement is possible only by using the specification of the selected name in which the prefix is the label of the block statement (i.e. `B_1.Sig_X`).

Example 8

```
type MEM_ARRAY is array ( 1 to 8 ) of BIT;
variable Mem_Var : MEM_ARRAY := "01000000";
Mem_Var(7)
```

The `Mem_Var` variable is the object of the `MEM_ARRAY` which is a one-dimensional array. That is why the expression list consists of only one value. The index indicates the array's element equal to '1'.

Example 9

```
variable MEM : BIT_VECTOR( 15 downto 0 );
MEM( 15 downto 8 )
```

The slice name presented in this example refers to the leftmost eight elements of the variable `MEM`.

Example 10

```
signal CLK : BIT;
CLK'EVENT -- An attribute EVENT of the signal CLK
```

The `'EVENT` attribute allows to check for an event that just occurred on the signal indicated by the prefix (in this example `CLK`).

Example 11

```
alias MULTI_VALUE_LOGIC_ALIAS : MULTI_VALUE_LOGIC_ARRAY is MULTI_VALUE_LOGIC_VECTOR;
MULTI_VALUE_LOGIC_ALIAS'RIGHT(1)
MULTI_VALUE_LOGIC_ALIAS'SIMPLE_NAME
```

Attribute `'RIGHT` indicates the right bound of the first range of `MVL_VECTOR` variable. However, `'SIMPLE_NAME` relates to the `MVL_ALIAS` alias.

Example 12

```
function "xor"( L, R : STD_ULOGIC ) return STD_ULOGIC;
attribute Built_In of "xor" : function is TRUE;
"xor" [STD_ULOGIC, STD_ULOGIC return STD_ULOGIC]'Built_In
```

The signature appearing in the attribute name specification allows describing the right function which has the `Built_In` attribute and not the standard one.

Important Notes

- Names must follow the rules of writing identifiers. In particular they must start with a letter and be composed of numbers and underscores. See identifier for details.
- None of the VHDL reserved words may be used as a name for other items.

Next Statement

Formal Definition

The next statement is used to complete execution of one of the iterations of an enclosing loop statement. The completion is conditional if the statement includes a condition.

Complete description: Language Reference Manual IEEE 1076-1993 § 8.10.

Simplified Syntax

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

Description

The `next` statement allows skipping a part of an iteration loop. If the condition specified after the `when` reserved word is TRUE, or if there is no condition at all, then the statement is executed. This results in skipping all statements below it until the end of the loop and passing the control to the first statement in the next iteration (Example 1).

The `next` statement may specify the label of the loop it is expected to influence. If no loop label is specified, then the statement applies to the innermost enclosing loop (Example 2)

Examples

Example 1

```
Loop_Z: for count_value in 1 to 8 loop
  Assign_1: A( count_value ) := '0';
  next when condition_1;
  Assign_2: A( count_value + 8 ) := '0';
end loop;
```

If the `condition_1` in the iteration `count_value` is TRUE, then the next statement will be executed. The next statement to be executed will be `Assign_1` in the iteration `count_value+1`. Otherwise, the sequence of operations is as specified, i.e. `Assign_2` is executed.

Example 2

```
Loop_X: for count_value in 1 to 8 loop
  Assign_1: A( count_value ) := '0';
  k := 0;
  Loop_Y: G loop
    Assign_2: B( k ) := '0';
    next Loop_X when condition_1;
    Assign_3: B( k + 8 ) := '0';
    k := k + 1;
  end loop Loop_Y;
end loop Loop_X;
```

If `condition_1` is TRUE, then the next statement is executed and the control goes to the assignment statement labeled `Assign_1` in the next iteration of `Loop_X`. If not, then the iteration is continued with `Assign_3` incrementing `k`.

Important Notes

- The next statement is often confused with the exit statement. The difference between the two is that the exit statement "exits" the loop entirely, while the next statement skips to the "next" loop iteration (in other words, it "exits" the current iteration of the loop).

Null Statement

Formal Definition

A statement that does not perform any action.

Complete description: Language Reference Manual IEEE 1076-1993 § 8.13

Simplified Syntax

```
null;
```

Description

The null statement does not perform any action and its only function is to pass on to the next statement. It can be used to indicate that when some conditions are met no action is to be performed. Such an application is useful in particular in conjunction with case statements to exclude some conditions (see examples).

Examples

```
case OPCODE is
  when "001" => Temporary_Data := Register_A and Register_B;
  when "010" => Temporary_Data := Register_A or Register_B;
  when "100" => Temporary_Data := not Register_A;
  when others => null;
end case;
```

The example shows an operand detection of a processor, restricted to some simple logical operations performed on registers. All other operations are blocked ("if the OPCODE is other than the explicitly listed, do nothing").

Important Notes

- The keyword `null` is used not only for "no operation" statements. It has a special meaning for variables of access types ("pointing at no object", which is the default value for such variables - see access type and allocator). There is also a null transaction in waveforms. These three applications of the keyword `null` should not be confused.

Operator Overloading

Definition:

Operator overloading is a declaration of a function whose designator is an operator symbol.

Full description: Language Reference Manual section § 2.3.1.

Simplified Syntax

```
function "operator" ( parameters ) return type;
function "operator" ( parameters ) return type is
  [ declarations ]
begin
  sequential statements
end [ function ];
```

Description

The operator is called overloaded if there is more than one function specifying it for different data and result types. VHDL allows defining operators of the same names as predefined operators, but for different operand types. Both can exist together in one specification, offering greater versatility to the user.

Such functions can be invoked both with prefix notation (Example 1) and usual infix notation (Example 2).

In the case when the operator is called and the operand values belong to different types, it is necessary to use the type conversion or the qualified expression in order to select appropriate operator (Example 3).

Examples

Example 1

```
type Log4 is ( '0', '1', 'Z', 'X' );
function "nand"( Left, Right : Log4 ) return Log4;
function "or"( Left, Right : Log4 ) return Log4;
signal S1, S2: Log4;
S1 <= "or"( '1', 'Z' );
S2 <= "nand"( S1, 'X' );
```

Functions `or` and `nand` implement basic logic operations for operands of type `Log4` overloading the predefined operations of the same names. In the above example these functions are called using the standard call syntax function (prefix).

Example 2

```
signal S3, S4: Log4;
S3 <= ( S1 nand S2 ) or 'X';
```

The operators or and nand are overloaded through the declarations as in the Example 1. Here the overloaded operators are used in the infix form, i.e. with the operator name between the operands.

Example 3

```
function "or"( Left, Right : Log4 ) return BIT;
signal S4: BIT;
S4 <= log4('1') or Log4('0');
```

The or operator is used here in a 4-value logical expression by connecting '1' and '0' operands with the or symbol. The qualified expression was used here to indicate the type of operands (which otherwise would be considered to be of the type BIT).

Important Notes

- Operators "+" and "-" can be defined both as binary operators (with two operands) or unary operators (with one operand).
- Invoking a user-defined overloaded operator always requires evaluation of both operands before the operation is executed (some predefined operators do not evaluate the right operand if the result can be decided from the left operand only).

Operators

Definition

Operators are means for constructing expressions.

Full description: Language Reference Manual section § 7.2.

Syntax

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

Description

VHDL has a wide set of different operators, which can be divided into groups of the same precedence level (priority). The table below lists operators grouped according to priority level, highest priority first.

Table 1. Operator priority

miscellaneous operators	**** abs not**
multiplying operators	* / mod rem
sign operators	+ -
adding operators	+ - &
shift operators	sll srl sla sra rol ror
relational operators	= /= < <= > > =>
logical operators	and or nand nor xor xnor

The expressions are evaluated form left to right, operations with higher precedence are evaluated first. If the order should be different from the one resulting from this rule, parentheses can be used (Example 1).

The operands, connected with each other by an operator, are evaluated before the operation described by that operator is carried out. For some operators the right operand is evaluated only when the left operand has a certain value assigned to it. The logical operators such as and, or, nand, nor defined for the BIT and BOOLEAN operands belong to those operators.

The operators for the predefined types are defined in the STANDARD package in the STD library. These operators are functions, which always return the same value when they are called with the same values of the actual parameters. These functions are called the pure function (see Function for details).

LOGICAL OPERATORS

The logical operators and, or, nand, nor, xor, xnor and not are defined for BIT and BOOLEAN types, as well as for one-dimensional arrays containing the elements of BIT and BOOLEAN. All these operators have the lowest priority, except for the operator not, which has the highest priority. The results of the logical operators for the predefined types are presented in the tables 2 through 8. The BIT type is represented by the values '0' and '1', while the BOOLEAN type by TRUE and FALSE.

Table 2. Operator not

A : BOOLEAN	A : BIT	not A : BOOLEAN	not A : BIT
TRUE	'1'	FALSE	'0'
FALSE	'0'	TRUE	'1'

Table 3. Operator and

A : BOOLEAN	A : BIT	B : BOOLEAN	B : BIT	A and B : BOOLEAN	A and B : BIT
TRUE	'1'	TRUE	'1'	TRUE	'1'
TRUE	'1'	FALSE	'0'	FALSE	'0'
FALSE	'0'	TRUE	'1'	FALSE	'0'
FALSE	'0'	FALSE	'0'	FALSE	'0'

Table 4. Operator or

A : BOOLEAN	A : BIT	B : BOOLEAN	B : BIT	A or B : BOOLEAN	A or B : BIT
TRUE	'1'	TRUE	'1'	TRUE	'1'
TRUE	'1'	FALSE	'0'	TRUE	'1'
FALSE	'0'	TRUE	'1'	TRUE	'1'
FALSE	'0'	FALSE	'0'	FALSE	'0'

Table 5. Operator xor

A : BOOLEAN	A : BIT	B : BOOLEAN	B : BIT	A xor B : BOOLEAN	A xor B : BIT
TRUE	'1'	TRUE	'1'	FALSE	'0'
TRUE	'1'	FALSE	'0'	TRUE	'1'
FALSE	'0'	TRUE	'1'	TRUE	'1'
FALSE	'0'	FALSE	'0'	FALSE	'0'

Table 6. Operator nand

A : BOOLEAN	A : BIT	B : BOOLEAN	B : BIT	A nand B : BOOLEAN	A nand B : BIT
TRUE	'1'	TRUE	'1'	FALSE	'0'
TRUE	'1'	FALSE	'0'	TRUE	'1'
FALSE	'0'	TRUE	'1'	TRUE	'1'
FALSE	'0'	FALSE	'0'	TRUE	'1'

Table 7. Operator nor

A : BOOLEAN	A : BIT	B : BOOLEAN	B : BIT	A nor B : BOOLEAN	A nor B : BIT
TRUE	'1'	TRUE	'1'	FALSE	'0'
TRUE	'1'	FALSE	'0'	FALSE	'0'
FALSE	'0'	TRUE	'1'	FALSE	'0'
FALSE	'0'	FALSE	'0'	TRUE	'1'

Table 8. Operator xnor

A : BOOLEAN	A : BIT	B : BOOLEAN	B : BIT	A xnor B : BOOLEAN	A xnor B : BIT
TRUE	'1'	TRUE	'1'	TRUE	'1'
TRUE	'1'	FALSE	'0'	FALSE	'0'
FALSE	'0'	TRUE	'1'	FALSE	'0'
FALSE	'0'	FALSE	'0'	TRUE	'1'

RELATIONAL OPERATORS

The relational operators allow checking relation between operands, i.e. to state whether they are equal, not equal or are ordered in a way defined by operator (Table 9). Both operands must be of the same type, and the result received is always of the Boolean type.

Table 9. Relational operations

=	Equality
/=	Inequality
<	Ordering "less than"
<=	Ordering "less than or equal"
>	Ordering "greater than"
>=	Ordering "greater than or equal"

The operators: equality and inequality are predefined for all types available in the language except the file type. For other relations the operands must be of a scalar type or one-dimensional array types.

The equality operator returns the value TRUE only when both operands have the same values, and FALSE when the values are different. The inequality operator returns the value TRUE when the operators are different and FALSE when they are equal. There are certain rules that are used to compare operands depending on their type: in case of the scalar type, the operand values are equal only when the values are the same. Two values of the composite type are equal only when each value of the left operand corresponds to the value of the right operand and vice versa. In the record the corresponding elements have identical identifiers, and in the array the corresponding elements are those which appear at the same positions of arrays. In particular two null arrays of the same type are always equal.

The operators: <, <=, >, and >= return the TRUE logical value only when the condition in the given relation is met, otherwise the FALSE value is returned (Example 4).

SHIFT OPERATORS

The shift operators are defined for the one-dimensional array with the elements of the type BIT or BOOLEAN. For the shift operator an array is the left operand L and integer is the right operand R. The right operand represents the number of positions the left operand should be shifted. As the result of shifting, the value of the same type as the left operand is returned. Table 10 below shows predefined shift operators.

Table 10. Shift operators

sll	Shift left logical
srl	Shift right logical
sla	Shift left arithmetic
sra	Shift right arithmetic
rol	Rotate left logical
ror	Rotate right logical

The operator sll returns the value of the L left operand, after it has been shifted R number of times. If R is equal to 0 or L is the null array, the left operand L is returned. The single left logical operation replaces L with concatenation of the rightmost (L'Length - 1) elements of L and a single value T'Left, where T is the element type of L. If R > the single shift operation is repeated R number of times. If R is a negative number, the value of the expression L srl -R is returned.

The operator srl returns the value of the left operand L after it has been shifted to right R times. In case when R is equal to 0 or L is the null array, the left operand L is returned. The single shifting operation replaces the operand L with the concatenation of the leftmost L'Length -1 elements of L and a single value T'Left, where T is the element type of L. If R is a negative number, then the value of expression L sll -R is returned, otherwise the single shift operation is repeated R number of times.

The operator sla returns the value of the left operand L after it has been shifted to the left R number of times. In case when R is equal to 0 or L is the null array the left operand L is returned. The single shift operation replaces L with concatenation of the rightmost L'Length -1 elements of L and a single value L'Right. If R is a negative number, the value of the expression L sra -R is returned, otherwise the single shift operation is repeated R number of times.

The operator sra returns the value of the left operand L after it has been shifted to the right R number of times. In case when R is equal to 0 or L is the null array, the left operand L is returned. The single shift operation replaces L with a value, which is the result of concatenation whose left argument is the leftmost L'Length -1 elements of L and whose right argument is L'Left. If R is a negative number, the value of the expression L sla -R is returned, otherwise the single shift operation is repeated R number of times.

The operator rol returns the value of the L left operand after it has been rotated to the left R times. In case when R is equal to 0 or L is the null array the left operand L is returned. The single shift operation replaces L with a value which is the result of concatenation of the rightmost L'Length -1 elements of L and a single value L'Left. If R > 1 the single shift operation is repeated R times. If R > 1 is a negative number, the expression L ror -R is returned.

The operator ror returns the value of the L left operand after it has been rotated to the right R number of times. In case when R is equal to 0 or L is the null array, the left operand L is returned. The single shift operation replaces L with a value which is the result of concatenation whose left argument is the leftmost L'Length -1 elements of L and whose right argument is L'Right. If R > 1 the single shift operation is repeated R times. If R is a negative number, the expression L rol -R is returned.

ADDING OPERATORS

Adding operators consist of addition, subtraction and concatenation. The adding operators are shown in the Table 11 below.

Table 11. Adding operators

+	Addition
-	Subtraction
&	Concatenation

The adding and subtraction operators perform mathematical operations, and their operands can be of any numeric type.

The concatenation (&) operator is defined for elements of one-dimensional arrays. In the concatenation, the following situations can take place:

- When both operands are one-dimensional arrays of the same type, the concatenation connects the two arrays into one. The new array contains the elements from both arrays. The direction of the new array is the same as the direction in the array of the left operand, but if the left operand is the null array then it is the same as the direction of the right operand. In case when both operands are null arrays, the direction of the right operand is assumed (Example 6).
- When one of the operands is a one-dimensional array and the second operand is a scalar of the same type as the elements of that array, then the result of the concatenation is the same as in the first point. However, in this case the second operand is treated as a one-dimensional array which contains only one element (Example 6).
- In case when both operands are of the same scalar type, then the result of concatenation is one-dimensional array with elements of the same types as the operands (Example 6).

SIGN OPERATORS

Sign operators are unary operators, i.e. have only one, right operand, which must be of a numeric type. The result of the expression evaluation is of the same type as the operand. There are two sign operators (Table 12).

Table 12. Sign operators

+	Identity
-	Negation

When (+) sign operator is used, the operand is returned unchanged, but In case of (-) sign operator the value of operand with the negated sign is returned. Because of the lower priority, the sign operator in the expression cannot be directly preceded by the multiplication operator, the exponentiation operator (**) or the abs and not operators. When these operators are used then sign operator and its operand should be enclosed in parentheses (Example 7).

MULTIPLYING OPERATORS

The multiplication and division operators are predefined for all integers, floating point numbers. Under certain conditions, they may be used for operations on physical type objects as well. The mod and rem operators, on the other hand, are defined only for the integers. When mod and rem operators are used, then both the operands and the result are of the same integer type. The multiplying operators are shown in the Table 13.

Table 13. Multiplying operators

*	Multiplication
/	Division
mod	Modulus
rem	Remainder

MISCELLANEOUS

The two miscellaneous operators are shown in the Table 14.

Table 14. Miscellaneous operators

**	Exponentiation
abs	Absolute value

The abs operator has only one operand. It allows defining the operand's absolute value. The result is of the same type as the operand.

The exponentiation operator has two operands. This operator is defined for any integer or floating point number. The right operand (exponent) must be of integer type. When the exponent is the positive integer, then the left operand is repeatedly multiplied by itself. When the exponent is the negative number, then the result is a reverse of exponentiation with the exponent equal to the absolute value of the right operand (Example 9). If the exponent is equal to 0 the result will be 1.

Examples

Example 1

```
v := a + y * x;
```

The multiplication y*x is carried out first, then a is added to the result of multiplication. This is because the multiplication operator has higher level of priority than the adding operator.

Example 2

```
variable We1, We2, We3, Wy : BIT := '1';
Wy := We1 and We2 xnor We1 nor We3;
```

For the initial value of the variables `We1`, `We2`, `We3` equal to '1', the result is assigned to the variable `Wy` and is equal to '0'.

Example 3

```
variable Zm1 : REAL := 100.0;
variable Zm2 : BIT_VECTOR( 7 downto 0 ) := ( '0', 0', 0', 0', 0', 0', 0', 0' );
variable Zm3, Zm4 : BIT_VECTOR( 1 to 0 );
Zm1 /= 342.54 -- TRUE
Zm1 = 100.0 -- TRUE
Zm2 /= "10000000" -- TRUE
Zm3 = Zm4 -- TRUE
```

Example 4

```
Zm1 > 42.54 -- TRUE
Zm1 >= 100.0 -- TRUE
Zm2 < ( '1', '0', '0', '0', '0', '0', '0', '0' ) -- TRUE
Zm3 <= Zm2 -- TRUE
```

Example 5

```
variable Zm5 : BIT_VECTOR( 3 downto 0 ) := "1011";
Zm5 sll 1 -- "0110"
Zm5 sll 3 -- "1000"
Zm5 sll -3 -- Zm5 srl 3
Zm5 srl 1 -- "0101"
Zm5 srl 3 -- "0001"
Zm5 srl -3 -- Zm5 sll 3
Zm5 sla 1 -- "0111"
Zm5 sla 3 -- "1111"
Zm5 sla -3 -- Zm5 sra 3
Zm5 sra 1 -- "1101"
Zm5 sra 3 -- "1111"
Zm5 sra -3 -- Zm5 sla 3
Zm5 rol 1 -- "0111"
Zm5 rol 3 -- "1101"
Zm5 rol -3 -- Zm5 ror 3
Zm5 ror 1 -- "1101"
Zm5 ror 3 -- "0111"
Zm5 ror -3 -- Zm5 rol 3
```

Example 6

```
constant B1: BIT_VECTOR := "0000"; -- four element array
constant B2: BIT_VECTOR := "1111"; -- four element array
constant B3: BIT_VECTOR := B1 & B2; -- eight element array, ascending range, value "00001111"
subtype BIT_VECTOR_TAB is BIT_VECTOR( 1 downto 0 );
```

```
constant B4: BIT_VECTOR_TAB := "01";
constant B5: BIT_VECTOR := B4 & B2; -- six element array, descending range, value "011111"
constant B6 : BIT := '0';
constant B7 : BIT_VECTOR := B2 & B6;-- five element array, ascending range, value "11110"
constant B8: BIT := '1';
constant B9: BIT_VECTOR := B6 & B8; -- two element array, ascending range, value "01"
```

Example 7

```
z := x * ( -y ) -- A legal expression
z := x / ( not y ) -- A legal expression
```

The same expressions without parentheses would be illegal.

Example 8

```
variable A, B :INTEGER;
variable C : REAL;
C := 12.34 * ( 234.4 / 43.89 );
A := B mod 2;
```

Example 9

```
2 ** 8 = 256
3.8 ** 3 = 54.872
4 ** ( -2 ) = 1 / ( 4**2 ) = 0.0625
```

Important Notes

- All predefined operators for standard types are declared in the package STANDARD.
- The operator not is classified as a miscellaneous operator only for the purpose of defining precedence. Otherwise, it is classified as a logical operator.

Package

Formal Definition

A package declaration defines the interface to a package.

Complete Description: Language Reference Manual § 2.5

Simplified Syntax

```
package package_name is
  package_declarations
end [ package ] [ package_name ];
```

Description

The package is a unit that groups various declarations, which can be shared among several designs. Packages are stored in libraries for greater convenience. A package consists of package declaration (mandatory) and may contain a single optional package body.

The purpose of a package is to declare shareable types, subtypes, constants, signals, files, aliases, component, attributes and groups. Once a package is defined, it can be used in multiple independent designs.

Items declared in a package declaration are visible in other design units if the use clause is applied (Example 1).

The two-part specification of a package (declaration and body) allows to declare the so-called deferred constants which have no value assigned in the package declaration (Example 2). The value for a deferred constant, however, must be declared in the package body accompanying the package declaration.

The VHDL Language Standard defines two standard packages, which must be available in any VHDL environment - package STANDARD and package TEXTIO. The former contains basic declarations of types, constants and operators, while the latter defines operations for manipulating text files. Both are located in the library STD. See respective topics for details.

Apart from the VHDL Language Standard there is another standard, which extends the language and supports the extensions in the form of a package: std_logic_1164.

Examples

Example 1

```
library Packages;
use Packages.AUXILIARY.all;

architecture STRUCT of Adder is
  -- ...
end architecture;
```

All declarations, which are inside the `AUXILIARY` package, may be used in the architecture body `STRUCT` of the design entity `Adder`. The package itself is stored in the library `Packages`.

Example 2

```
library IEEE;
use IEEE.std_logic_1164.all;

package AUXILIARY is
  type MUX_input is array ( INTEGER range <> ) of STD_LOGIC_VECTOR( 0 to 7 );
  type operation_set is ( SHIFT_LEFT, ADD );
  subtype MUX_address is POSITIVE;
  function Compute_Address( IN1 : MUX_input ) return MUX_address;
  constant Deferred_Con : INTEGER;
end;
```

Package `AUXILIARY` contains a function declaration and a deferred constant, thus a package body had to be declared for this package.

Important Notes

- Package declaration may contain a subprogram (function or procedure) declaration; subprogram body is not allowed here and must appear in the package body.
- Package body must accompany a package declaration if the declaration contains subprogram declarations or deferred constants.

Package Body

Formal Definition

A package body defines the bodies of subprograms and the values of deferred constants declared in the interface to the package.

Complete description: Language Reference Manual IEEE 1076-1993 § 2.6.

Simplified Syntax

```
package body package_name is
  [ package_body_declarations ]
  [ subprogram bodies declarations ]
  [ deferred constants declarations ]
end [ package body ] [ package_name ];
```

Description

The package body includes complete definitions of subprogram body declarations as well as values of deferred constants declared in corresponding package declarations. Other declarations (similar to those of package declaration) are also allowed here, but are visible only inside the package body.

The deferred constant, which has been declared in a package declaration, may be used before its full declaration only in a default expression for a local generic parameter, local port or formal parameter of subprogram.

Examples

Example 1

```
library IEEE;
use IEEE.std_logic_1164.all;

package AUXILIARY is
  type MUX_input is array ( INTEGER range <> ) of STD_LOGIC_VECTOR( 0 to 7 );
  type operation_set is ( SHIFT_LEFT, ADD );
  subtype MUX_Address is POSITIVE;
  function Compute_Address( IN1 : MUX_input ) return MUX_address;
  constant Deferred_Con : INTEGER;
end package;

package body AUXILIARY is
  function Compute_Address( IN1 : MUX_input ) return MUX_address is
  begin
    -- ...
  end function;
  constant Deferred_Con : INTEGER := 177;
end package body;
```

First, the package is specified here and then the accompanying package body. Note that both have the same name.

Important Notes

- Declarations other than values of deferred constants and subprogram bodies are not visible outside the package body and can be used only locally, inside it.
- Each package can have only one body.

Physical Type

Formal Definition

Physical type is a numeric scalar that represents some quantity. Each value of a physical type has a position number that is an integer value. Any value of a physical type is a straight multiple of the primary unit of measurement for that type.

Complete description: Language Reference Manual IEEE 1076-1993 § 3.1, § 3.1.3.

Simplified Syntax

```
type type_name is range left_bound to right_bound
units
  primary_unit_name
  secondary_unit_name = number primary_unit_name
  secondary_unit_name = number primary_unit_name
  ...
end units type_name

type type_name is range left_bound downto right_bound
units
  primary_unit_name
  secondary_unit_name = number primary_unit_name
  secondary_unit_name = number primary_unit_name
  ...
end units type_name
```

Description

A physical type allows to define measurement units for some physical quantity, like length, time, pressure, capacity, etc.

The range, specified at the beginning of a physical type, defines the minimum and maximum values of a given quantity expressed in the base units (see primary unit declaration below). The bounds of the range of a physical type should be in the form of locally static expression. The expression is classified as locally static if it is possible to determine its value without running the code. The value of an expression used as a range for a physical type must be of integer type. It is legal to use negative bounds.

At the core of a physical type declaration is the definition of a primary unit, and optionally, multiple secondary units. The primary unit serves as the base unit for representing values of the specified type. The secondary units are defined as multiplicity of primary units or previously specified secondary units. Their declarations may contain only integer literals.

Each value of a physical type has a corresponding position number, which is the number of primary units represented by that unit name. This way values specified in different units of the same type can be compared (Example 2).

For all physical types, both TIME and user-defined, relational and arithmetic operators can be used.

The VHDL standard predefines only one physical type: TIME, which is defined in the STANDARD package.

The value of one unit can be written in short form using the name of the unit only (i.e. without the value 1 - Example 3).

Examples

Example 1

```
type CAPACITY is range 0 to 1E5
units
  pF; -- picofarad
  nF = 1000 pF; -- nanofarad
  uF = 1000 nF; -- microfarad
  mF = 1000 uF; -- millifarad
  F = 1000 mF; -- farad
end units;
```

The primary unit in this example is one picofarad and the maximum value of the `CAPACITY` type object is 1E5 pF.

Example 2

```
type DISTANCE is range 0 to 1E5
  units
    um; -- micrometer
    mm = 1000 um; -- millimeter
    in_a = 25400 um; -- inch
  end units;
variable Dis1, Dis2 : DISTANCE;
Dis1 := 28 mm;
Dis2 := 2 in_a - 1 mm;
if Dis1 < Dis2 then -- ...
```

Both, a comparison and an arithmetic operation can be performed on visually different measurement units as long as they are defined as the same type with the same base unit.

Example 3

```
Some_Variable := mF;
```

This assignment means that the variable `Some_Variable` will be assigned the value of 1 mF.

Important Notes

- Physical types are not synthesizable. As a result, delays in signal assignments are not synthesizable as well.
- It is not allowed to use floating point values in physical type declarations, i.e. if a conversion from millimeters to inches (25.4 mm = 1 in) would have to be performed, then millimeters could not be used as the base unit.

Port

Formal Definition

A channel for dynamic communication between a block and its environment.

Complete description: Language Reference Manual IEEE 1076-1993 § 1.1.1.2

Simplified Syntax

```
port( port_declaration, port_declaration, ... );
-- port declarations:
port_signal_name : in port_signal_type := initial_value
port_signal_name : out port_signal_type := initial_value
port_signal_name : inout port_signal_type := initial_value
port_signal_name : buffer port_signal_type := initial_value
port_signal_name : linkage port_signal_type := initial_value
```

Description

Ports are a part of the block interface: external - if defined by a design entity, or internal - if defined by a block statement. Each element listed in a port interface list declares a formal port, which provides a channel for dynamic communication between a block and its environment.

In practice, ports are most often used in entities and components, where they serve for declaring interface signals of a design entity (system design) or component, respectively.

In both cases, each interface element is a signal. It can be preceded by a keyword `signal`. After the signal's name, a mode is specified. The mode declares the direction of data flow through the port. There are five modes available in VHDL for ports:

- in input port. A variable or a signal can read a value from a port of mode in, but is not allowed to assign a value to it.
- out output port. It is allowed to make signal assignments to a port of the mode out, but it is not legal to read from it.
- inout bi-directional port. Both assignments to such a port and reading from it are allowed.
- buffer output port with read capability. It differs from inout in that it can be updated by at most one source, whereas inout can be updated by zero or more sources.
- linkage. The value of the port may be read or updated, but only by appearing as an actual corresponding to an interface object of mode linkage.

If a port is declared with a reserved word `bus`, then the signal declared by that port is a guarded signal of signal kind `bus`.

A port can be assigned a default value, which is specified by an expression evaluating to the same type as the port itself.

Examples

Example 1

```
entity Mux8to1 is
port(
  Inputs : in STD_LOGIC_VECTOR( 7 downto 0 );
  Select_s : in STD_LOGIC_VECTOR( 2 downto 0 );
  Output : out STD_LOGIC );
end entity;
```

Entity of a multiplexor 8-to-1 contains three ports: eight data inputs (specified as a vector), address inputs and one output.

Example 2

```
component Memory_Device is
port(
  Data : inout STD_LOGIC_VECTOR( 7 downto 0 );
  Addr : in STD_LOGIC_VECTOR( 9 downto 0 );
  Not_Chip_Select : in STD_LOGIC;
  Read_Not_Write : in BIT );
end component;
```

Memory device is specified here as a component with four signals: data is a bi-directional data bus, address is a ten-bit input, and Not_Chip_Select and Read_Not_Write are single inputs signals. Note that the keyword `is` in the header can be used in VHDL 93 only (in VHDL 87 it must be omitted).

Important Notes

- Ports declarations are signal declarations and port signals need not to be re-declared.

Procedure

Formal Definition

A procedure is a subprogram that defines algorithm for computing values or exhibiting behavior. Procedure call is a statement.

Complete description: Language Reference Manual IEEE 1076-1993 § 2.1, § 2.2, § 8.5, § 9.3.

Simplified Syntax

```
procedure procedure_name ( formal_parameter_list )
procedure procedure_name ( formal_parameter_list ) is
  [ procedure_declarations ]
begin
  sequential statements
end [ procedure ] [ procedure_name ];
```

Description

The procedure is a form of subprograms. It contains local declarations and a sequence of statements. Procedure can be called in any place of the architecture. The procedure definition consists of two parts:

- the procedure declaration, which contains the procedure name and the parameter list required when the procedure is called;
- the procedure body, which consists of the local declarations and statements required to execute the procedure.

PROCEDURE DECLARATION

The procedure declaration consists of the procedure name and the formal parameter list.

In the procedure specification, the identifier and optional formal parameter list follow the reserved word `procedure` (Example 1).

Objects classes constants, variables, signals, and files can be used as formal parameters. The class of each parameter is specified by the appropriate reserved word, unless the default class can be assumed (see below). In case of constants, variables and signals, the parameter mode determines the direction of the information flow and it decides which formal parameters can be read or written inside the procedure. Parameters of the file type have no mode assigned.

There are three modes available: in, out, and inout. When in mode is declared and object class is not defined, then by default it is assumed that the object is a constant. In case of inout and out modes, the default class is variable. When a procedure is called, formal parameters are substituted by actual parameters. If a formal parameter is a constant, then actual parameter must be an expression. In case of formal parameters such as signal, variable and file, the actual parameters must be objects of the same class. Example 2 presents several procedure declarations with parameters of different classes and modes.

A procedure can be declared also without any parameters.

PROCEDURE BODY

Procedure body defines the procedure's algorithm composed of sequential statements. When the procedure is called it starts executing the sequence of statements declared inside the procedure body.

The procedure body consists of the subprogram declarative part after the reserved word `is` and the subprogram statement part placed between the reserved words `begin` and `end`. The key word procedure and the procedure name may optionally follow the `end` reserved word.

Declarations of a procedure are local to this declaration and can declare subprogram declarations, subprogram bodies, types, subtypes, constants, variables, files, aliases, attribute declarations, attribute specifications, use clauses, group templates and group declarations (Example 3).

A procedure can contain any sequential statements (including wait statements). A wait statement, however, cannot be used in procedures which are called from a process with a sensitivity list or from within a function. Example 4 and Example 5 present two sequential statements specifications.

PROCEDURE CALL

A procedure call is a sequential or concurrent statement, depending on where it is used. A sequential procedure call is executed whenever control reaches it, while a concurrent procedure call is activated whenever any of its parameters of in or inout mode changes its value.

All actual parameters in a procedure call must be of the same type as formal parameters they substitute.

OVERLOADED PROCEDURES

The overloaded procedures are procedures with the same name but with different number or different types of formal parameters. The actual parameters decide which overloaded procedure will be called (Example 6).

Examples

Example 1

```
procedure Procedure_1( variable X, Y : inout Real );
```

The above procedure declaration has two formal parameters: bi-directional variables X and Y of the REAL type.

Example 2

```
procedure Proc_1( constant In1 : in INTEGER; variable O1 : out INTEGER );
procedure Proc_2( signal Sig : inout BIT );
```

Procedure Proc_1 has two formal parameters: the first one is a constant and it is of mode in and of the INTEGER type, the second one is an output variable of the INTEGER type.

Procedure Proc_2 has only one parameter, which is a bi-directional signal of the type BIT.

Example 3

```
procedure Proc_3 ( X, : inout INTEGER ) is
  type Word_16 is range 0 to 65536;
  subtype Byte is Word_16 range> 0 to 255;
  variable Vb1, b2, b3 : REAL;
  constant Pi : Real := 3.14;
  procedure Compute ( variable V1, V2 : REAL )is
  begin
    -- subprogram_statement_part
  end procedure;
begin
  -- subprogram_statement_part
end procedure;
```

The example above present different declarations which may appear in the declarative part of a procedure.

Example 4

```
procedure Transcoder_1( variable Value : inout BIT_VECTOR( 0 to 7 ) ) is
begin
  case Value is
    when "00000000" => Value := "01010101";
    when "01010101" => Value := "00000000";
    when others => Value := "11111111";
  end case;
end procedure;
```

The procedure Transcoder_1 transforms the value of a single variable, which is therefore a bi-directional parameter.

Example 5

```
procedure Comp_3( In1, R : in REAL; Step : in INTEGER; W1, W2 : out REAL ) is
  variable counter: INTEGER;
begin
  W1 := 1.43 * In1;
  W2 := 1.0;
  L1: for counter in 1 to Step loop
    W2 := W2 * W1;
    exit L1 when W2 > R;
  end loop L1;
  assert ( W2 < R )
    report "Out of range"
      severity Error;
end procedure;
```

The Comp_3 procedure calculates two variables of mode out: W1 and W2, both of the REAL type. The parameters of mode in: In1 and R constants are of REAL type and Step of the INTEGER type. The W2 variable is calculated inside the loop statement. When the value of W2 variable is greater than R, the execution of the loop statement is terminated and the error report appears.

Example 6

```
procedure Calculate( W1, W2 : in REAL; signal Out1 : inout INTEGER );
procedure Calculate( W1, W2 : in INTEGER; signal Out1 : inout REAL );
-- calling of overloaded procedures:
Calculate( 23.76, 1.632, Sign1 );
Calculate( 23, 826, Sign2 );
```

The procedure `Calculate` is an overloaded procedure as the parameters can be of different types. Only when the procedure is called the simulator determines which version of the procedure should be used, depending on the actual parameters.

Important Notes

- The Procedure declaration is optional - procedure body can exist without it. If, however, a procedure declaration is used, then a procedure body must accompany it.
- Subprograms (procedures and functions) can be nested.
- Subprograms can be called recursively.
- Synthesis tools usually support procedures as long as they do not contain the wait statements.

Process Statement

Formal Definition

A process statement defines an independent sequential process representing the behavior of some portion of the design.

Complete description: Language Reference Manual IEEE 1076-1993 § 9.2.

Simplified Syntax

```
[ label: ] [ postponed ] process [ ( sensitivity_list ) ] [ is ]
  [ declarations ]
begin
  sequential_statements
end process [ label ];
```

Description

The process statement represents the behavior of some portion of the design. It consists of the sequential statements whose execution is made in order defined by the user.

Each process can be assigned an optional label.

The process declarative part defines local items for the process and may contain declarations of: subprograms, types, subtypes, constants, variables, files, aliases, attributes, use clauses and group declarations. It is not allowed to declare signals or shared variables inside processes.

The statements, which describe the behavior in a process, are executed sequentially, in the order in which the designer specifies them. The execution of statements, however, does not terminate with the last statement in the process, but is repeated in an infinite loop. The loop can be suspended and resumed with wait statements. When the next statement to be executed is a wait statement, the process suspends its execution until a condition supporting the wait statement is met. See respective topics for details.

A process declaration may contain optional sensitivity list. The list contains identifiers of signals to which the process is sensitive. A change of a value of any of those signals causes the suspended process to resume. A sensitivity list is a full equivalent of a wait on sensitivity_list statement at the end of the process. It is not allowed, however, to use wait statements and sensitivity list in the same process. In addition, if a process with a sensitivity list calls a procedure, then the procedure cannot contain any wait statements.

Examples

Example 1

```
entity D_FF is
port(
  D, LK : in BIT;
  Q : out BIT := '0';
  NQ : out BIT := '1' );
end entity;

architecture A_RS_FF of D_FF is
begin
  BIN_P_RS_FF: process ( CLK )
  begin
    if rising_edge( CLK ) then
      Q <= D;
      NQ <= not D;
    end if;
  end process;
end architecture;
```

The flip-flop has two input ports: `D`, `CLK` and two output ports: `Q` and `NQ`. The value of the input signal `D` is assigned to the output `Q` when the value of the `CLK` changes from '0' to '1'. The change of the signal value initiates the process `BIN_P_RS_FF`, because the signal `CLK` is on the sensitivity list of this process. The change of the `CLK` value from the logical zero to the logical one is sensed by the use of the `if` statement and `rising_edge` function.

Important Notes

- Sensitivity list and explicit wait statements may not be specified in the same process.

Range

Formal Definition

A specified subset of values of a scalar type.

Complete description: Language Reference Manual IEEE 1076-1993 § 3.1.

Simplified Syntax

```
range left_bound to right_bound
range left_bound downto right_bound
range <>
```

Description

The range specifies a subset of values of a scalar type. This range can be null range if the set contains no values. A range can be either ascending or descending.

A range is called ascending if it is specified with the keyword `to` as the direction and the left bound value is smaller than the right bound (otherwise the range is null). A range is descending if the range is specified with the keyword `downto` as the direction and the left bound is greater than the right bound (otherwise the range is null).

A value X belongs to a range if this range is not a null range and the following relation holds:

lower bound of the range <= X <= upper bound of the range

A range can be undefined. Such a range is used in declaration of unconstrained arrays and is illustrated in Example 2.

Examples

Example 1

```
1 to 100
7 downto 0
5 to 0
```

The first range is an ascending range of the values of INTEGER type. The second range is also of INTEGER type, but descending. Finally, the third range is `null`.

Example 2

```
type Mem is array ( NATURAL range <> ) of BIT_VECTOR( 7 downto 0 );
```

The type `Mem` is declared as an unconstrained array of bytes (8-bit wide vectors of bits). Note the way an undefined range is declared.

Important Notes

- Ranges do not have to be bounded by discrete numbers. Enumeration types can also be used for ranges (for example 'a' to 'z').

Record

Formal Definition

<Q A composite type whose values consist of named elements.>

Complete description: Language Reference Manual IEEE 1076-1993 § 3.2.2.

Simplified Syntax

```
type record_type_name is
record
  element_name : element type;
  { element_name : element type; }
end record [ record_type_name ];
```

Description

The record type allows declaring composite objects whose elements can be of different types. This is the main difference from arrays, which must have all elements of the same type. All elements are declared with individual names together with subtype indication. If two or more elements are of the same subtype they can be declared together (Example 1). The names of elements in each record must be distinct. The same element name, however, can be used in different records.

The value of an object of type record is a composite value, consisting of the values of its elements. The assignment of a value to an object of the type record can be realized either through an aggregate or through individual assignments to elements (selected names).

Aggregate-based assignment to a record, either positional or named association, can be used (Example 2). If the positional association is used, it is assumed that the elements are listed in the order defined in the record declaration. If the others choice is used, it must represent at least one element. If there are two or more elements assigned by the others choice, then all these elements have to be of the same type.

When individual assignment to elements are used then each element id referenced by the record object name followed by a dot and element's name (Example 3).

Expression assigned to an element of a record must result in a value of the same type as the element.

Examples

Example 1

```
type Register_Name is ( AX, BX, CX, DX );
type Operation is record
  Mnemonic : STRING( 1 to 10 );
  OpCode : BIT_VECTOR( 3 downto 0 );
  Op1, Op2, Res : Register_Name;
end record;
```

The record type defined above represents an information from an instruction list of a processor. There are five elements here: mnemonic code (a string), operation code (four bits), two operands and the destination. Note that the last three elements are declared together as they are of the same type.

Example 2

```
-- type declarations are given in L<"Example 1">
variable Instr1, Instr2: Operation;
-- ...
Instr1 := ( "ADD AX, BX", "0001", AX, BX, AX );
Instr2 := ( "ADD AX, BX", "0010", others => BX );
```

Here, the two assignments to variables of the record type (`Operation`) are performed with aggregates. Note the way the choice `others` was used in the second example.

Example 3

```
-- type declarations are given in L<"Example 1">
variable Instr3 : Operation;
-- ...
Instr3.Mnemonic := "MUL AX, BX";
Instr3.Op1 := AX;
```

In this case direct assignments to individual elements of a record object are performed. Note the way an element is referenced: record name, dot, and element name.

Important Notes

- Linear records (i.e. record, where elements are of not of composite type) are generally synthesizable.
- Files are not allowed as elements of records.

Report Statement

Formal Definition

A statement that displays a message.

Complete description: Language Reference Manual IEEE 1076-1993 § 8.3.

Simplified Syntax

```
[ label: ] report expression [ severity severity_level ] ;
```

Description

The report statement is very much similar to Assertion Statement. The main difference is that the message is displayed unconditionally. Its main purpose is to help in the debugging process.

The expression specified in the `report` clause must be of predefined type `STRING`, and it is a message that will be reported when the assertion violation occurred.

If the severity clause is present, it must specify an expression of predefined type `SEVERITY_LEVEL`, which determines the severity level of the assertion violation. The `SEVERITY_LEVEL` type is specified in the STANDARD package and contains following values: NOTE, WARNING, ERROR, and FAILURE. If the severity clause is omitted in a report statement it is implicitly assumed to be NOTE (unlike the Assertion Statement, where the default is ERROR).

Examples

Example 1

```
while counter <= 100 loop
  if counter > 50 then
    report "The counter is over 50.";
  end if;
  -- ...
end loop;
```

Whatever happens inside the loop, if the value of counter is greater than 50 it will be reported by the listed message. The severity clause is omitted here because the selected level is the same as the default one (NOTE).

Important Notes

- The report statement was introduced as late as in VHDL 93 and is equivalent to the `assert false` statement. The latter form was the only acceptable in VHDL 87.

Reserved Word

Definition:

The reserved word is an identifier reserved in the VHDL language for a special purpose.

Complete description: Language Reference Manual IEEE 1076-1993 § 13.9.

Description

The reserved words cannot be used as explicitly declared identifiers. The complete list of reserved words is given below:

abs	exit	nor	select
after	file	not	severity
alias	for	null	signal
all	function	of	shared
and	generate	on	sla
architecture	generic	open	sll
array	group	or	sra
assert	guarded	others	srl
attribute	if	out	subtype
begin	impure	package	then
block	in	port	to
body	inertial	postponed	transport
buffer	inout	procedure	type
bus	is	process	unaffected
case	label	pure	units
component	library	range	until
configuration	linkage	record	use
constant	literal	register	variable
disconnect	loop	reject	wait
downto	map	rem	when
else	mod	report	while
elsif	nand	return	with

end	new	rol	xnor
entity	next	ror	xor

Important Notes

- VHDL is case insensitive, therefore there is no difference using either uppercase or lowercase for reserved words.
- If an identifier is placed between leading and trailing backslashes, it becomes an extended identifier and is no longer a reserved word (e.g. \port \ is not a reserved word).

Resolution Function

Formal Definition

A resolution function is a function that defines how the values of multiple sources of a given signal are to be resolved into a single value for that signal.

Complete description: Language Reference Manual IEEE 1076-1993 section § 2.4.

Simplified Syntax

```
function function_name ( parameters ) return type;
function function_name ( parameters ) return type is
  [ declarations ]
begin
  sequential statements
end [ function ] [ function_name ];
```

Description

The resolution function allows multiple values to drive a single signal at the same time. This is particularly important for buses, which are connecting multiple sources of data.

The specification of a resolution function is the same as for ordinary functions with one requirement: the resolution function must be pure.

Resolution functions are associated with signals that require resolution by including the name of the resolution function in the declaration of signals or in the declaration of the signal subtype.

Examples

Example 1

```
type STD_ULOGIC is ( 'U', -- Uninitialized
                     'X', -- Forcing Unknown
                     '0', -- Forcing 0
                     '1', -- Forcing 1
                     'Z', -- High Impedance
                     'W', -- Weak Unknown
                     'L', -- Weak 0
                     'H', -- Weak 1
                     '-' -- Don't care
                     );
type STD_ULOGIC_VECTOR is array (NATURAL range <>) of STD_ULOGIC;
function resolved (s : STD_ULOGIC_VECTOR) return STD_ULOGIC;
subtype STD_LOGIC is resolved STD_ULOGIC;
type STD_LOGIC_VECTOR is array (NATURAL range <>) of STD_LOGIC;

type stdlogic_table is array(STD_ULOGIC, STD_ULOGIC) of STD_ULOGIC;
constant resolution_table : stdlogic_table := (
  -- ---------------------------------------------------------
  -- | U X 0 1 Z W L H - | |
  -- ---------------------------------------------------------
           ('U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U'), -- | U |
           ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'), -- | X |
           ('U', 'X', '0', 'X', '0', '0', '0', '0', 'X'), -- | 0 |
           ('U', 'X', 'X', '1', '1', '1', '1', '1', 'X'), -- | 1 |
           ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X'), -- | Z |
           ('U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X'), -- | W |
           ('U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X'), -- | L |
           ('U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X'), -- | H |
           ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X') -- | - |
           );
function resolved (s : STD_ULOGIC_VECTOR) return STD_ULOGIC is
  variable result : STD_ULOGIC := 'Z'; -- weakest state default
begin
  -- the test for a single driver is essential otherwise the
  -- loop would return 'X' for a single driver of '-' and that
  -- would conflict with the value of a single driver unresolved
  -- signal.
  if (s'length = 1) then return s(s'low);
  else
    for i in s'range loop
      result := resolution_table(result, s(i));
    end loop;
  end if;
  return result;
end function resolved;
```

The example is a part of the std_logic_1164 Package specification. The name of the resolution function called `Resolved` is included into the declaration of the subtype `STD_LOGIC`. The resolution function itself is declared at the end of the example.

Important Notes

- Standard types (BIT and BIT_VECTOR) are not resolved and it is not possible to specify multiple-source buses with these types. This is quite restrictive for typical applications, which use buses.
- Because STD_LOGIC and STD_LOGIC_VECTOR are resolved and can handle buses, they became the de facto industrial standard types.

Resume

Definition:

The action of a wait statement when the conditions for which the wait statement is waiting are satisfied.

Complete description: Language Reference Manual IEEE 1076-1993 § 12.6.3.

Description

A suspended process (i.e. a process waiting for a condition specified in a wait statement to be met) is resumed when the condition is met. The execution of resumed process is started immediately in the current simulation cycle (time), unless the process is not postponed. In the latter case, the process execution is postponed to the last simulation cycle at the current simulation time.

A resumed process executes its statements sequentially in a loop until a wait statement is encountered. When this happens, the process becomes suspended again.

Examples

Example 1

```
process ( CLK, RST )
begin
  if RST='1' then
    Q <= '0';
  elsif rising_edge( CLK ) then
    Q <= D;
  end if;
end process;
```

In this Example 1 of a D flip-flop, the process is sensitive to the two signals: `CLK` and `RST`. It will resume when any of the two signals will change its value. Resuming of the process will cause the execution of the 'if' statement (which his the only one statement in this process) and then the process will suspend again, waiting for a change on either `RST` or `CLK`.

Important Notes

- A resumed process not necessarily executes all its statements: if there are multiple wait statements the execution suspends on the next 'wait'.

Return Statement

Formal Definition

The return statement is used to complete the execution of the innermost enclosing function or procedure body.

Complete description: Language Reference Manual IEEE 1076-1993 § 8.12.

Simplified Syntax

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

Description

The return statement ends the execution of a subprogram (Function or Procedure) in which it appears. It causes an unconditional jump to the end of a subprogram (Example 1).

If a return statement appears inside nested subprograms it applies to the innermost subprogram (i.e. the jump is performed to the next `end function` or `end procedure` clause).

This statement can only be used in a procedure or function body. The return statement in a procedure may not return any value, while a return in a function must return a value (an expression) which is of the same type as specified in the function after the `return` keyword (Example 2).

Examples

Example 1

```
procedure RS( signal S, R: in BIT; signal Q, NQ: inout BIT ) is
begin
  if ( S = '1' and R = '1' ) then
    report "forbidden state: S and R are equal to '1'";
    return;
  else
    Q <= S and NQ;
    NQ <= R and Q;
  end if;
end procedure;
```

The return statement located in the if then clause causes the procedure to terminate when both S and R are equal to '1'. The procedure would terminate even if the end if would be followed by some other statements.

Example 2

```
P1: process
  type real_NEW is range 0.0 to 1000.0;
  variable a, b : real_NEW := 2.0;
  variable c : REAL;
  function Add( Oper_1, Oper_2: real_NEW ) return REAL is
    variable result : REAL;
  begin
    result := REAL(Oper_1) + REAL(Oper_2);
    return result;
  end function;
begin
  c := Add( a, b);
end process;
```

The return statement in a function must return a value of the type specified in the function header after the return clause.

Important Notes

- Although the return statement is a sequential one, it is not allowed to use it in a process.

Scalar Type

Formal Definition

Scalar type is a type whose values have no elements. Scalar types consist of enumeration types, integer types, physical types, and floating point types. Enumeration types and integer types are called discrete types. Integer types, floating point types, and physical types are called numeric types. All scalar types are ordered; that is, all relational operators are predefined for their values.

Complete description: Language Reference Manual IEEE 1076-1993 § 3.1.

Syntax

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

Description

The scalar type values cannot contain any composite elements. All values in a specific scalar type are ordered. Due to this feature all relational operators are predefined for those types. Also, each value of a discrete or physical type has a position number.

All numeric types (i.e. integer, floating point, and physical) can be specified with a range that constrains the set of possible values.

Please refer to respective topics for more information on different scalar types (enumeration type, integer type, floating point type, physical type).

Sensitivity List

Formal Definition

A list of signals a process is sensitive to.

Complete description: Language Reference Manual IEEE 1076-1993 § 9.2.

Simplified Syntax

```
( signal_name, signal_name, ... )
```

Description

The sensitivity list is a compact way of specifying the set of signals, events on which may resume a process. A sensitivity list is specified right after the keyword `process` (Example 1).

The sensitivity list is equivalent to the wait on statement, which is the last statement of the process statement section.

Only static signal names, for which reading is permitted, may appear in the sensitivity list of a process, i.e. no function calls are allowed in the list.

Examples

Example 1

```
DFF : process ( CLK, RST )
begin
  if RST= '1' then
    Q <= '0';
  elsif rising_edge( CLK ) then
    Q <= D;
  end if;
end process;

-- DFF : process
-- begin
-- if RST= '1' then
-- Q <= '0';
-- elsif rising_edge( CLK ) then
-- Q <= D;
-- end if;
-- wait on RST, CLK;
-- end process;
```

Here, the process is sensitive to the `RST` and `CLK` signals, i.e. an event on any of these signals will cause the process to resume. This process is equivalent to the one described in the comment section.

Important Notes

- A process with a sensitivity list may not contain any explicit wait statements. Also, if such a process statement is a parent of a procedure, then that procedure may not contain a wait statement as well.

Signal Assignment

Formal Definition

A signal assignment statement modifies the projected output waveforms contained in the drivers of one or more signals

Complete description: Language Reference Manual IEEE 1076-1993 § 8.4, § 9.5

Simplified Syntax

```
signal_name <= [delay_mechanism ] waveform;

signal_name <= [delay_mechanism ] waveform_1 when condition_1 else
  waveform_2 when condition_2 else
  ...
  waveform_n;

with selection select
  signal_name <= [delay_mechanism ] waveform_1 when choice1,
  waveform_2 when choice2,
  ...
  waveform_n when others;
```

Description

Signal assignment statement can appear inside a process or directly in an architecture. Accordingly, sequential signal assignment statements and concurrent signal assignment statements can be distinguished. The latter can be divided into simple concurrent signal assignment, conditional signal assignment and selected signal assignment.

The target signal can be either a name (simple, selected, indexed, or slice) or an aggregate.

All signal assignments can be delayed. See delay for details.

Sequential Signal Assignment

If a sequential signal assignment appears inside a process, it takes effect when the process suspends. If there are more than one assignments to the same signal in a process before suspension, then only the last one is valid. Regardless of the number of assignments to a signal in a process, there is always only one driver for each signal in a process (Example 1).

If a signal is assigned a value in a process and the signal is on the sensitivity list of this process, then a change of the value of this signal may cause reactivation of the process (Example 2).

Concurrent Signal Assignment

The concurrent signal assignment statements can appear inside an architecture. Concurrent signal assignments are activated whenever any of the signals in the associated waveforms change their value. Activation of a concurrent signal assignment is independent from other statements in given architecture and is performed concurrently to other active statements (Example 3). If there are multiple assignments to the same signal then multiple drivers will be created for it. In such a case, the type of the signal must be of the resolved type (see Resolution Function).

Conditional Signal Assignment

Conditional signal assignment is a form of a concurrent signal assignment and plays the same role in architecture as the if then else construct inside processes. A signal is assigned a waveform if the Boolean condition supported after the `when` keyword is met. Otherwise, the next condition after the `else` clause is checked, etc. Conditions may overlap.

A conditional signal assignment must end with an unconditional else expression (Example 4).

Selected Signal Assignment

Selected signal assignment is a concurrent equivalent of a sequential case construct. All choices for the expression must be included, unless the others clause is used as the last choice (Example 5). Ranges and selections can be used as the choice (Example 6). Choices are not allowed to overlap.

Examples

Example 1

```
signal A, B, C, X, Y, Z : INTEGER;
process ( A, B, C )
begin
  X <= A + 1;
  Y <= A * B;
  Z <= C - X;
  Y <= B;
end process;
```

When this process is executed, signal assignment statements are performed sequentially, but the second assignment (`Y <= A * B`) will never be executed because only the last assignment to Y will be activated. Moreover, in the assignment to Z only the previous value of X will be used as the `A + 1` assignment will take place when the process suspends.

Example 2

```
signal A, B, C, X, Y, Z : INTEGER;
process ( A, B, C )
begin
```

```
  X <= A + 1;
  Y <= A * B;
  Z <= C - X;
  B <= Z * C;
end process;
```

When the process is activated by an event on the signal C this will cause change on the signal B inside a process, which will in turn reactivate the process because B is in its sensitivity list.

Example 3

```
architecture Concurrent of Half_Adder is
begin
  Sum <= A xor B;
  Carry <= A and B;
end architecture;
```

The above architecture specifies a half adder. Whenever A or B changes its value, both signal assignments will be activated concurrently and new values will be assigned to Sum and Carry.

Example 4

```
architecture Conditional of TriState_Buffer is
begin
  Buffer_Output <= Buffer_Input when Enable= '1'
    else 'Z';
end architecture;
```

The architecture specifies a tri-state buffer. The buffer output Buffer_Output will be assigned the value of buffer input Buffer_Input only when the Enable input is active high. In all other cases the output will be assigned high impedance state.

Example 5

```
architecture Concurrent of Universal_Gate is
begin
  with Command select
  DataOut <= InA and InB when "000",
  InA or InB when "001",
  InA nand InB when "010",
  InA nor InB when "011",
  InA xor InB when "100",
  InA xnor InB when "101",
  'Z' when others;
end architecture Concurrent;
```

Architecture of Universal_Gate is specified with a selected signal assignment. Depending on the value of the Command signal, the DataOut signal will be assigned value resulting from the logical operation of two inputs. If none of the specified codes appears, the output is set to high impedance.

Example 6

```
with Int_Command select
  Mux_Output <= InA when 0 | 1,
    InB when 2 to 5,
    InC when 6,
    InD when 7,
    'Z' when others;
```

A specialized multiplexer is defined here with a selected signal assignment. Note that both range and selections can be used as a choice.

Important Notes

- Signal assignment statements are generally synthesizable but delays are usually ignored.
- Choices in selected signal assignment are separated by colons.
- All signal assignments can be labeled for improved readability.

Signal Declaration

Formal Definition

Signal is an object with a past history of values. A signal may have multiple drivers, each with a current value and projected future values. The term signal refers to objects declared by signal declarations and port declarations.

Complete description: Language Reference Manual § 4.3.1.2.

Simplified Syntax

```
signal signal_name : type [signal_kind] [ := expression ];
```

Description

Signals are the primary objects describing a hardware system and are equivalent to "wires". They represent communication channels among concurrent statements of system's specification. Signals and associated mechanisms of VHDL (like signal assignment statements, resolution function, delays, etc.) are used to model inherent hardware features such as concurrency, inertial character of signal propagation, buses with multiple driving sources, etc. Each signal has a history of values and may have multiple drivers, each of which has a current value and projected future values. All signal parameters are accessible by means of signal attributes.

Signals can be explicitly declared in the declarative part of:

- package declaration; signals declared in a package are visible in all design entities using the package (through the use clause);
- architecture (see Architecture); such signals are visible inside the architecture only;
- block (see Block Statement); the scope of such signals is limited to the block itself;
- entity declaration (see Entity); signals declared in the entity declarative part are common to all design entities whose interfaces are defined by the given entity declaration.

Moreover, a port declaration in an entity is an implicit signal declaration (Example 1). A signal declared this way is visible in all architectures assigned to that entity.

A signal declaration contains one or more identifiers (i.e. more than one signal can be declared in one statement) and a subtype indicator. Each signal name is an identifier and creates one separate signal. The (sub)type in the signal declaration can be of any scalar or composite type. Optionally, it may have some constraints. File and access types are not allowed for signals. Some typical signal declarations are given in the Example 1, below.

A signal can be assigned an initial (default) value in its declaration. It the value is produced by an expression, it must be of the same type as the signal itself. If there is no expression given in the signal declaration, then the default value of the signal is the left bound of the specified type (see Example 2).

A signal may be declared with a signal_kind statement, which can be either a register or bus. Such signal must be of a resolved type. A register type signal retains its current value even when all its drivers are turned off. However, the signal_kind bus relies on the resolution function to supply a "no-drive" value (see Resolution Function for details).

Examples

Example 1

```
library IEEE;
use IEEE.std_logic_1164.all;
entity Data_Transmitter is
  port( Data : STD_LOGIC_VECTOR( 15 downto 0 ) );
end entity;

architecture Example_Declaration of Data_Transmitter is
  signal Temp : STD_LOGIC;
  signal Flag_C, Flag_Z : BIT;
begin
  -- ...
end architecture;
```

Each statement of the architecture `Example_Declaration` may use any of the four signals:

- `Data` (16-bit vector), declared as a port in the entity part (above the architecture section),
- Temp that is a single signal of the STD_LOGIC type,
- `Flag_C` a single BIT signal
- `Flag_Z` another single BIT signal

Note that the signals `Flag_C` and `Flag_Z` are declared together in the same line because they both are of the same type.

Example 2

```
type Four_VL is ( 'X', 0', 1', Z' );
signal Sig1 : Four_VL;
signal Sig2 : Four_VL := 'X';
signal Sig3 : Four_VL := '0';
```

All three above listed signals are of the same type, but their default values are specified in different ways. Sig1 will be assigned the leftmost value of the type, i.e. 'X' (leftmost item in the first line), `Sig2` is explicitly assigned the same value. However, as this is the leftmost value of the signal type in this assignment, it is redundant and can be omitted. Finally, `Sig3` is assigned the '0' value. Since '0' is not the leftmost value of the type, it has to be assigned explicitly to the signal.

Important Notes

- It is illegal to declare signals in a process or a subprogram (except as formal parameters).
- Each port specified in an entity is accessible as a signal in every architecture assigned to this entity and need not to be declared again.
- A signal may never be of a file or access type.
- Despite that value assignment to a signal is made with the '<=' symbol, it is not applicable to the default value listed in the signal declaration, where the ':=' symbol must be used.
- If a signal is to be driven by more than one source (i.e. it will be assigned values in more than one statement), it has to be declared as of resolved type (see Resolution Function and Driver).
- The signal_kinds (register and bus) are not supported by synthesis tools.

Slice

Formal Definition

A one-dimensional array of a sequence of consecutive elements of another one-dimensional array.

Complete description: Language Reference Manual IEEE 1076-1993 § 6.5.

Simplified Syntax

```
object_name ( discrete_range )
function_call ( discrete_range )
```

Description

The slice is a part of a one-dimensional array, from a single element up to complete array.

The prefix used for a slice is the name of the parent array.

The index used for a slice must fall in the range of the indexes of the parent array. Moreover, the direction of the slice indexes must be the same as the direction of indexes of parent array (either ascending or descending).

The slice is an object which can be used in the same way as its parent array: if the parent array is a signal, then any its slice is also a signal, etc.

If the discrete range of a slice is null then the slice is null as well.

Examples

Example 1

```
signal Data_Bus : BIT_VECTOR( 31 downto 0 ); -- parent array
Data_Bus( 31 downto 26 ) -- slice 1
Data_Bus( 24 downto 24 ) -- slice 2
Data_Bus( 24 downto 30 ) -- slice 3
Data_Bus( 15 to 31 ) -- no slice - ERROR!
```

The first slice is a 6-element subarray of the `Data_Bus`. The second slice contains one element. Slice 3 is a null slice (the range is null). Finally, the fourth example is an error due to different directions of the parent array and the slice.

Important Notes

- The direction of the parent array and its slice must match (i.e. in both cases either `to` or `downto` keyword must be used).

Standard Package

Definition

The STANDARD package predefines a number of types, subtypes, and functions which are visible to all design units.

Complete definition: Language Reference Manual § 14.2.

Description

The STANDARD package is a part of the Language Specification. It defines basic types, subtypes, and functions, together with operators available for each of the (sub)types defined. The operators are specified implicitly. Below is a complete list of declared types, together with their predefined operators.

Contents:

The STANDARD package declares following types:

- BOOLEAN (with predefined operators "and", "or", "nand", "nor", "xor", "xnor", "not", "=", "/=", "<", "<=", ">", ">="),
- BIT (with predefined operators "and", "or", "nand", "nor", "xor", "xnor", "not", "=", "/=", "<", "<=", ">", ">="),
- CHARACTER (with predefined operators "=", "/=", "<", "<=", ">", ">="),
- SEVERITY_LEVEL (with predefined operators "=", "/=", "<", "<=", ">", ">="),
- INTEGER (with predefined operators "=", "/=", "<", "<=", ">", ">=", "+", "-", "abs", "*", "/", "mod", "rem", "**"),
- REAL (with predefined operators "=", "/=", "<", "<=", ">", ">=", "+", "-", "abs", "*", "/", "**"),
- TIME (with predefined operators "=", "/=", "<", "<=", ">", ">=", "+", "-", "abs", "*", "/"),
- STRING (with predefined operators "=", "/=", "<", "<=", ">", ">=", "&"),
- BIT_VECTOR (with predefined operators "and", "or", "nand", "nor", "xor", "xnor", "not", "sll", "srl", "sla", "sra", "rol", "ror", "=", "/=", "<", " <=", ">", ">=", "&"),
- FILE_OPEN_KIND (with predefined operators "=", "/=", "<", "<=", ">", ">="),
- FILE_OPEN_STATUS (with predefined operators "=", "/=", "<", "<=", ">", ">="),

and three subtypes:

- DELAY_LENGTH (subtype of TIME),
- POSITIVE (subtype of INTEGER),
- NATURAL (subtype of INTEGER).

See BOOLEAN, Concatenation, CHARACTER, INTEGER Type, REAL Type, Physical Type, STRING, BIT_VECTOR and File Type for details on respective types.

Important Notes

- Use of the STANDARD package is implicitly assumed by every VHDL simulator and compiler and need not to be explicitly declared by the `use` clause.
- The user may not modify the contents of the package.

STD_LOGIC

Definition

A nine-value resolved logic type `STD_LOGIC` is not a part of the VHDL Standard in the context of IEEE 1076-1993 Language Reference Manual. It is defined in IEEE Std 1164-1993 IEEE Standard Multivalue Logic System for VHDL Model Interoperability. The IEEE Std 1076-2008 incorporates and enhances the content of IEEE 1164-1993, thus making `STD_LOGIC`, derived types, operators, functions, and procedures a part of VHDL standard.

Syntax

```
type STD_ULOGIC is ( 'U', -- Uninitialized
                     'X', -- Forcing Unknown
                     '0', -- Forcing 0
                     '1', -- Forcing 1
                     'Z', -- High Impedance
                     'W', -- Weak Unknown
                     'L', -- Weak 0
                     'H', -- Weak 1
                     '-' -- Don't Care
                     );
function resolved ( s : STD_ULOGIC_VECTOR ) return STD_ULOGIC;
subtype STD_LOGIC is resolved STD_ULOGIC;
```

Description

The `STD_ULOGIC` type is an extension of the standard BIT type. It defines nine values, which allow specifying logical systems. Like `BIT`, this type is not resolved, i.e. it is not allowed to specify two simultaneous value assignments to a signal of the STD_ULOGIC type.

In order to facilitate specification of multiple-driven signals (like data buses) the std_logic_1164 Package defines resolution function for STD_ULOGIC, which in turn serves as a basis for declaration of `STD_LOGIC` type.

The std_logic_1164 package defines overloaded logical operators ("and", "nand", "or", "nor", "xor", and "not") for operands of the `STD_ULOGIC` type. Moreover, two conversion functions are defined as well: `STD_ULOGIC` to `BIT` (function To_Bit), and `BIT` to `STD_ULOGIC` (function To_StdULogic).

Examples

Example 1

```
signal Flag_C : STD_LOGIC := 'Z';

ALU : process
begin
  -- ...
  if Carry then
    Flag_C <= '1';
  end if;
end process;

Comm : process
begin
  -- ...
  Flag_C <= '0';
end process;
```

`STD_LOGIC` is a resolved type, which means that multiple assignments to the same object are legal. If `Flag_C` was of the `STD_ULOGIC` type, such a code would not be acceptable.

Important Notes

- `STD_LOGIC` is defined as a subtype of `STD_ULOGIC`, therefore all operators and functions defined for `STD_ULOGIC` can be applied to `STD_LOGIC`.
- `STD_LOGIC` is the industry standard logic type and in practice majority of signals are of this type (or its vector derivative, STD_LOGIC_VECTOR type).

std_logic_1164 Package

Definition

Package std_logic_1164 is NOT a part of the VHDL Standard Definition. It is defined as IEEE Std 1164.

Description

The std_logic_1164 Package contains definitions of types, subtypes, and functions, which extend the VHDL into a multi-value logic. It is not a part of the VHDL Standard, but it is a separate Standard of the same standardization body (Institute of Electrical and Electronics Engineers, IEEE).

Main reason for development and standardization of std_logic_1164 was the need for more logical values (than the two defined by the type BIT in the STANDARD package) with resolution function. The types STD_LOGIC and STD_LOGIC_VECTOR (declared in std_logic_1164 package) became de facto industrial standards.

Contents:

The package contains the following declarations:

- type STD_ULOGIC: unresolved logic type of 9 values;
- type STD_ULOGIC_VECTOR: vector of STD_ULOGIC;
- function resolved resolving a STD_ULOGIC_VECTOR into STD_ULOGIC;
- subtype STD_LOGIC as a resolved version of STD_ULOGIC;
- type STD_LOGIC_VECTOR: vector of STD_LOGIC;
- subtypes X01, X01Z, UX01, UX01Z subtypes of resolved STD_ULOGIC containing the values listed in the names of subtypes (i.e. UX01 contains values 'U', 'X', '0', and '1', etc.);
- logical functions for STD_LOGIC, STD_ULOGIC, STD_LOGIC_VECTOR and STD_ULOGIC_VECTOR;
- conversion functions between STD_ULOGIC and bit, STD_ULOGIC and BIT_VECTOR, STD_LOGIC_VECTOR and BIT_VECTOR and vice-versa;
- functions rising_edge and falling_edge for edge detection of signals.
- x-value detection functions, is_x, which detect values 'U', 'X', 'Z', 'W', '-' in the actual parameter.

See STD_LOGIC and STD_LOGIC_VECTOR for details.

Important Notes

- The std_logic_1164 Package is copyrighted and may not be altered (either by modifying/removing existing declarations or adding new ones).
- In order to use any of the declarations of the std_logic_1164 package, the 'library' and 'use' clauses have to be used:

```
library IEEE;
use IEEE.std_logic_1164.all;
```

STD_LOGIC_VECTOR

Definition:

The STD_LOGIC_VECTOR type is predefined in the std_logic_1164 package as a standard one-dimensional array type with each element being of the STD_LOGIC type.

STD_LOGIC_VECTOR is not a part of the VHDL Standard. Instead, it is defined by IEEE Std 1164

Syntax

```
type STD_LOGIC_VECTOR is array ( NATURAL range <> ) of STD_LOGIC;
```

Description

STD_LOGIC_VECTOR is an unconstrained (unbound) vector of resolved nine-value logic elements (STD_LOGIC type), which are defined in the std_logic_1164 package.

The std_logic_1164 package defines overloaded logical operators ("and", "nand", "or", "nor", "xor", and "not") for operands of the STD_LOGIC_VECTOR type. In addition, conversion functions from and to BIT_VECTOR are supported as well.

Assignment to an object of the STD_LOGIC_VECTOR type can be performed in the same way as in case of arrays, i.e. using single element assignments, concatenation, aggregates, slices or any combination of them. Moreover, because the elements are of resolved type it is allowed to make multiple assignments to a STD_LOGIC_VECTOR object type. In such a case, the resolution function defined for STD_LOGIC is used.

Examples

Example 1

```
type T_Data is array ( 7 downto 0 ) of STD_LOGIC;
signal Data_Bus, Memory : T_Data;

CPU : process
  variable Register_A : T_Data;
begin
  -- ...
  Data_Bus <= Register_A;
end process;

Mem : process
begin
  -- ...
  Data_Bus <= Memory;
end process;
```

STD_LOGIC_VECTOR is the best choice for buses, which are driven from different places, like the above listed data bus. Such a multiple assignment would be illegal if a BIT_VECTOR was used.

Important Notes

- STD_LOGIC_VECTOR should not be confused with the STD_ULOGIC_VECTOR type. Elements of the latter are of the type STD_ULOGIC and are unresloved version of STD_LOGIC. This means that it is illegal for the two values (e.g. '0' and 'Z') to be simultaneously driven into a signal of the STD_ULOGIC type.
- In order to use the STD_LOGIC_VECTOR type, the std_logic_1164 package must be explicitly listed at the beginning of an entity:

  ```
  library IEEE; use IEEE.std_logic_1164.all;
  ```

STRING

Formal Definition

The string type is predefined in the package Standard as a standard one-dimensional array type with each element being of the type Character.

Complete description: Language Reference Manual IEEE 1076-1993, § 3.2.1.2.

Syntax

```
type STRING is array ( POSITIVE range <> ) of CHARACTER;
```

Description

The type string is an unconstrained vector of elements of the type Character. The size of a particular vector must be specified during its declaration (see Example 1). The way the vector elements are indexed depends on the defined range - either ascending or descending (see Range).

Assignment to an object of the type string can be performed in the same way as in case of any arrays, i.e. using single element assignments, concatenation, aggregates, slices or any combination of them.

The package Standard contains declarations of the predefined operators for the type String: "=", "/=", "<", "<=", ">", ">=" and "&". Relational operators allow to compare two strings, while the concatenation operator allows to concatenate two strings, a string and a character and two characters to create a string.

Examples

Example 1

```
constant Message1 : STRING( 1 to 19 ) := "hold time violation";
signal Letter1 : CHARACTER;
signal Message2 : STRING( 1 to 10 );
-- ...
Message2 <= "Not" & Letter1;
```

Important Notes

- Unlike BIT_VECTOR, where the value of index is of the type NATURAL (from 0 up to INTEGER'High), the index of STRING has a POSITIVE value, being an integer greater than 0. It would be an error, then, to declare a STRING with a range with zero as one of the boundary values.
- Strings are written in double quotes. Single elements, however, are of the type CHARACTER, therefore values assigned to single elements (referred by the index) are specified in single quotes.
- Strings play supplementary role for system modeling as they do not reflect any particular feature of hardware. They are used mostly for issuing messages during simulation. (See Assertion Statement and

Report Statement.)

Subtype

Formal Definition

A type together with a constraint.

A value belongs to a subtype of a given type if it belongs to the type and satisfies the constraint; the given type is called the base type of the subtype. A type is a subtype of itself. Such a subtype is said to be unconstrained because it corresponds to a condition that imposes no restriction.

Complete description: Language Reference Manual IEEE 1076-1993 § 4.2.

Simplified Syntax

```
subtype subtype_name is base_type range range_constraint;
```

Description

Subtype distinguishes a subset of values of some type.

The part of the subtype declaration is the subtype indication, which denotes some other type or subtype. The type_mark in a subtype_indication must refer to a type or a subtype that was declared earlier (Example 1).

The constraints given in the subtype indication must correspond to the subtype. For scalar types - range constrains can be applied, for arrays - index constraints are applicable. Records cannot have any constraints. Access type may have index type constraints only when their type_mark denotes an array type. If the subtype declaration does not contain any constraints then the subtype is the same as the (sub)type denoted by the type_mark.

A special form of the subtype indication may include a resolution function name (Example 2). This form is not allowed for declarations of access and file subtypes.

There are two predefined subtypes specified in the package STANDARD: NATURAL and POSITIVE. Both are subtypes of the type INTEGER. The package std_logic_1164 also contains declarations of subtypes, which are constrained subtypes of the STD_LOGIC: X01, X01Z, UX01, and UX01Z.

Examples

Example 1

```
subtype DIGITS is INTEGER range 0 to 9;
```

INTEGER is a predefined type and the subtype DIGITS will constrain the type to ten values only, reducing the size of registers if the specification is synthesized.

Example 2

```
function RESOLVE_VALUE ( anonymous: BIT_VECTOR ) return BIT;
subtype BIT_NEW is RESOLVE_VALUE BIT;
```

The subtype BIT_NEW is a resolved version of the type BIT due to the reference to a resolution function RESOLVE_VALUE specified earlier.

Important Notes

- A subtype declaration does not define a new type.
- A subtype is the same type as its base type; thus, no type conversion is needed when objects of a subtype and its base type are assigned (in either direction). Also, the set of operations allowed on operands of a subtype is the same as the set of operations on its base type.
- Using subtypes of enumerated and integer types for synthesis is strongly recommended as synthesis tools infer an appropriate number of bits in synthesized registers, depending on the range.

Suspend

Formal Definition

Suspend is a process that stops executing and waits either for an event or for a time period to elapse.

Complete description: Language Reference Manual IEEE 1076-1993 § 8.1, § 12.6.4.

Description

When a wait statement is encountered in a process, the process becomes suspended, i.e. it stops its execution until the condition supported by the wait statement is met. Depending upon the type of a wait statement there can be several conditions for resuming (continuing execution of) a suspended process:

- timeout specified has expired (`wait for` statement)
- a logical condition is met (`wait until` statement)
- an event on a signal took place (`wait on` statement)

See Wait Statement for details.

Examples

Example 1

```
wait until CLK'event and CLK = '0';
```

A process containing such a wait statement will be suspended until a falling edge on the `CLK` signal will be encountered.

Important Notes

- If no condition is specified in a wait statement, the process suspends forever.

Testbench

Complete Definition:

Testbench is not defined by the VHDL Language Reference Manual and has no formal definition.

Simplified Syntax

```
entity testbench_entity is
end entity;

architecture testbench_architecture of testbench_entity is
  signal declarations
  component declarations
begin
  component instantiations
  stimuli ( test vectors )
end architecture;
```

Description

The testbench is a specification in VHDL that plays the role of a complete simulation environment for the analyzed system (unit under test, UUT). A testbench contains both the UUT as well as stimuli for the simulation.

The UUT is instantiated as a component of the testbench and the architecture of the testbench specifies stimuli for the UUT's ports, usually as waveforms assigned to all output and bidirectional ports of the UUT.

The entity of a testbench does not have any ports as this serves as an environment for the UUT. All the simulation results are reported using the assert and report statements.

Examples

Example 1

```
entity Test_Decoder_bcd is
end entity;

architecture Struct_1 of Test_Decoder_bcd is
  component Decoder_bcd is
    port(
      enable : in BIT;
      led : in STD_ULOGIC_VECTOR( 3 downto 0 );
      bcd : out BIT_VECTOR( 1 downto 0 ) );
  end component;
  signal bcd : BIT_VECTOR( 1 downto 0 ) := "11";
  signal Enable : BIT := '1';
  signal led : STD_ULOGIC_VECTOR( 3 downto 0 );
begin
  U1: Decoder_bcd port map( Enable, led, bcd );
  bcd <= "00" after 5 ns,
    "01" after 15 ns,
    "10" after 25 ns,
    "11" after 35 ns;
  assert ( bcd = "00" ) and ( led = "0001")
      or ( bcd = "01" ) and ( led = "0010" )
      or ( bcd = "10" ) and ( led = "0100" )
      or ( bcd = "11" ) and ( led = "1000" )
    report "There is an incorrect value on the output led"
      severity error;
end architecture;
```

The design entity `Test_Decoder_BCD` is designed to verify correctness of the `Decoder_BCD`. This testbench applies stimuli to the `bcd` inputs and when the value of the `led` signal is other than a single '1' on the position corresponding to the binary value of the `bcd` signal, with all other bits equal to zero, the listed error is reported.

Important Notes

- Testbenches should allow automated verification of the UUT, with reports on success or failure of each sub-test.
- In case of sequential units under test, a clock signal should be supported in the testbench. Typically, it is realized as a separate process in the testbench architecture.
- In order to stop the simulation with a testbench, stimuli are often specified inside a process which contains a non-conditional wait statement at the end; such statement suspends the execution of the testbench forever.

Type

Formal Definition

A set of values and a set of operations.

Complete description: Language Reference Manual IEEE 1076-1993 § 3, § 4.1.

Simplified Syntax

```
type type_name is type_definition;
type type_name;
```

Description

Each object in VHDL has to be of some type, which defines possible values and operations that can be performed on this object (and other objects of the same type). The set of operations of a type consists of:

- explicitly declared subprograms that have a parameter or result of the particular type; such subprograms can be either predefined (in standard packages) or user-defined;
- basic operations (assignments, allocators, selected names, indexed names, slice names)
- numeric literals, literal null, string literal, bit string literals, aggregates or predefined attributes - depending on particular type.

There are four classes of types in VHDL:

- scalar types (values of these types have no elements),
- composite types (values of these types consist of element values),
- access types (provide access to objects of a given type) and
- files (provide access to objects that contain a sequence of values of a given type).

See respective topics for details.

Apart from predefined types (available through the packages STANDARD and std_logic_1164), the user can define his/hers own types. A user-defined type can be of any of the four classes mentioned above. The rules for defining types are described in detail in the corresponding topics.

Important Notes

- A type defines not only a set of values, but also a set of operators.
- VHDL is strongly typed language which causes that two types defined in exactly the same way (i.e. lexically identical) but differing only by names will be considered different.
- If a translation from one type to another is required, then type conversion must be applied, even if the two types are very similar (like assigning a natural variable to an integer variable).

Type Conversion

Formal Definition

An expression that converts the value of a subexpression from one type to the designated type of the type conversion.

Complete description: Language Reference Manual IEEE 1076-1993 § 7.3.5.

Simplified Syntax

```
type_mark ( expression )
```

Description

VHDL is a strongly typed language. This causes that objects of even closely related types need a conversion, if they are supposed to be used together.

The result of the conversion is an object, which type is the same as the type specified by the type_mark (Example 1). The type of the operand (the expression) must be known independently from the context. Moreover, the operand cannot be an aggregate, an allocator, the literal null, or a string literal.

A type conversion is restricted to closely related types, i.e. must conform to the following rules::

- All abstract numeric types (integers and floating point numbers) are closely related types. When a floating-point number is converted into an integer, the result is rounded to the nearest integer.
- Two arrays are closely related types if:
    - Both arrays have the same dimensions,
    - their elements are of the same type,
    - for each range, the index types are the same or closely related.
- No other types are closely related.

If the type_mark indicates an unrestricted array type, then after the conversion the range boundaries are moved from the operand. If the type_mark indicates a restricted array type, then the range boundaries of the array are described based on this type. After the conversion, the elements values of the array are equal to the elements values before the conversion.

Examples

Example 1

```
variable Data_Calculated, Param_Calculated : INTEGER;
-- ...
Data_Calculated := INTEGER( 74.94 * REAL( Param_Calculated ) );
```

This assignment contains two type conversions: first `Param_Calculated` is converted to a REAL value in order to multiply it with a universal real. Then the result of the multiplication is converted back to an INTEGER and assigned to `Data_Calculated`.

Important Notes

- No conversion is needed between any type or subtype defined on its basis.
- Two arrays may be closely related even if corresponding index positions have different directions.

Use Clause

Formal Definition

Achieves direct visibility of declarations that are visible by selection.

Complete description: Language Reference Manual IEEE 1076-1993 § 10.4.

Simplified Syntax

```
use library_name.package_name.item;
use library_name.package_name;
use library_name.package_name.all;
```

Description

The use clause makes visible items specified as suffixes in selected names listed in the clause. In practice, the use clause makes visible declarations specified in packages and has the following form:

```
use library_name.package_name.item;
```

If a designer wants to have all declarations in a package visible, then the 'item' clause should be substituted by the reserved word `all`.

The use clause is valid for the design unit immediately following it and for all secondary design units assigned to this design unit (if it is a primary design unit).

Examples

```
library IEEE;
use IEEE.std_logic_1164.all;

library IEEE;
use IEEE.std_logic_1164.STD_ULOGIC;
use IEEE.std_logic_1164.rising_edge;
```

In the first example, all declarations specified in the package std_logic_1164 (that belongs to the library IEEE) have been made visible.

The second example makes visible the `rising_edge` function, which is declared in the same package. The function uses the type STD_ULOGIC, therefore declaration of this type is also made visible.

Important Notes

- Using multiple value logic and resolution functions requires using library clause and use clause like in the first example.

Variable Assignment

Formal Definition

A variable assignment statement replaces the current value of a variable with a new value specified by an expression.

Complete description: Language Reference Manual IEEE 1076-1993 § 8.5

Simplified Syntax

```
variable_name := expression;
```

Description

The variable assignment statement modifies the value of the variable. The new value of the variable is obtained by assigning an expression to this variable. In order to distinguish variable assignment from signal assignment, the variable assignment symbol is different (:=).

The expression assigned to a variable must give results of the same type as the variable. The target at the left-hand side of the assignment can be either a name of a variable or an aggregate.

In the first case, the target can be in the form of simple name, selected name, indexed name or slice name (Example 1).

In case of aggregate as the target of the assignment, the type of the aggregate must be determinable from the context, including the fact that the type must be of the composite type. Each element of the aggregate must be in the form of the locally static name, which represents variable (Example 2).

The element association of aggregate, similarly to names, may have forms that are more complex: selected name, indexed name or slice name (Example 3).

Examples

Example 1

```
variable X, Y : REAL;
variable A, B : BIT_VECTOR( 0 to 7 );
type BIT_RECORD is record
  bit_field : BIT;
  int_field : INTEGER;
end record;
variable C, D : BIT_RECORD;
X := 1000.0;
A := B;
A := "11111111";
A( 3 to 6 ) := ( '1', 1', 1', 1' );
A( 0 to 5 ) := B ( 2 to 7 );
```

```
A(7) := '0';
B(0) := A(6);
C.bit_field := '1';
D.int_field := C.int_field;
```

The above examples of variable assignments are grouped in the following way: after the declarations the first group is a group of assignments with simple names as targets, then slice names, indexed names and finally selected names.

Example 2

```
variable E : BIT;
variable I : INTEGER;
( E, I ) := C;
```

The aggregate used above as a target for a variable assignment could be used for the variable `C` declared in such a way as in the Example 1. Variable `E` will be assigned the value of `C.bit_field` and `I` - `C.int_field`.

Example 3

```
type BIT_VECTOR_RECORD is record
  a: BIT_VECTOR( 0 to 7 );
  b: INTEGER;
end record;
variable G, H : BIT_VECTOR_RECORD;
( C.bit_field, C.int_field ) := D; -- aggregate with selected name
( G.a( 0 to 7 ), K ) := H; -- aggregate with sliced name
( G.a( 0 ), K ) := D; -- aggregate with indexed name
```

Aggregates can use different forms of names.

Important Notes

- Variable assignment can be labeled.
- Variable assignment takes effect immediately.
- Variable assignment cannot be specified with a delay.

Variable Declaration

Formal Definition

Variable is an object with a single current value.

Complete description: Language Reference Manual IEEE 1076-1993 § 4.3.1.3.

Simplified Syntax

```
[ shared ] variable variable_name : type [ := expression ];
```

Description

Variables are objects which store information local to processes and subprograms (procedures and functions) in which they are defined. Their values can be changed during simulation through the variable assignment statements.

A variable declaration includes one or more identifiers, a subtype indication and an optional globally static expression defining the initial value for the variable(s). The identifiers specify names for the variables, with one identifier per each variable. The variable can be declared to be of any type or subtype available, either constrained or unconstrained (Example 1).

The initial value of a variable can be assigned by a globally static expression. The expression must reference a value of the same type as the variable itself. If the variable is declared to be of a composite type other than a string, `BIT_VECTOR` or `STD_LOGIC_VECTOR`, then an aggregate must be used (see Example 2).

If the initial value of a variable is not specified, then it is implicitly set as the left bound of the type used. For example for a scalar named T, the default initial value will be T'LEFT. For a composite type, the initial value will be the aggregate consisting of the set of the default values of all the scalar elements. The default initial value for a variable of the access type is null.

Variables declared in processes are initialized with their default values, given either explicitly or implicitly, at the start of the simulation. Variables declared in subprograms are initialized each time the subprogram is called.

The scope of variables is limited to the process or subprogram they are defined in. The only exception to this rule is a shared variable, which may be shared by multiple processes.

Variables can be also declared outside of a procedure or process to be shared between many processes. Shared variables may be declared within an architecture, block, generate statement or package. Declaration of a shared variable must be preceded by the `shared` keyword (Example 3).

Although the Language Reference Manual allows several processes to access a single shared variable it does not define what happens when two or more conflicting processes try to access the same variable at the same time. Such a situation may lead to unpredictable results and therefore should be avoided.

Examples

Example 1

```
type Mem is array ( NATURAL range <>, NATURAL range <> ) of STD_LOGIC;
variable Delay1, Delay2 : TIME;
variable RAM1: Mem ( 0 to 1023, 0 to 8 );
```

The type `Mem` is specified as an unconstrained memory (the limit depends on the implementation of the NATURAL subtype). The variable `RAM1`, specified in the third line, is based on the `Mem` type and is defined as a subtype constrained to a 1 KB memory.

Both `Delay1` and `Delay2` variables are of the TIME type and are declared in the second line.

Example 2

```
type Mem is array ( NATURAL range <>, NATURAL range <> ) of STD_LOGIC;
variable Temporary_Condition : BOOLEAN : = TRUE;
variable RAM2 : Mem ( 0 to 7, 0 to 7 ) := (
  ( '0', '0', '0', '0', '0', '0', '0', '0' ),
  ( '0', '0', '0', '0', '0', '0', '0', '0' ),
  ( '0', '0', '0', '0', '0', '0', '0', '0' ),
  ( '0', '0', '0', '0', '0', '0', '0', '0' ),
  ( '0', '0', '0', '0', '0', '0', '0', '0' ),
  ( '0', '0', '0', '0', '0', '0', '0', '0' ),
  ( '0', '0', '0', '0', '0', '0', '0', '0' ),
  ( '0', '0', '0', '0', '0', '0', '0', '0' ) );
```

Both variables are initialized according to the types used. Note the aggregate has been used for initializing `RAM2`, which is an 8x8 bit memory.

Example 3

```
shared variable Free_Access : BOOLEAN := TRUE;
```

The shared variable `Free_Access` statement is used to determine whether a shared resource can be accessed at any particular time. However, in real applications a signal declaration would better serve for this purpose because using signals makes the specification more deterministic.

Important Notes

- Unlike signals, variables have neither history nor future, because according to its definition, each variable has only current value. No checking for the last event, time elapsed since the last event, previous value, etc. can be performed on variables.
- The non-shared variables are limited to subprograms and processes only.
- If a value of a variable is read before it is assigned in a clocked process (i.e. where operations are performed when an edge of clock signal is detected) then a register will be synthesized for this variable. A similar situation inside a combinatorial process may lead to generation of a latch.
- Variables declared in a subprogram are synthesized as combinatorial logic.

- During simulation variables consume a lot less (even an order of magnitude) memory space than signals. Therefore, it is highly recommended to use them for storage elements (such as memories) instead of signals.
- Shared variables may cause a system to be non-deterministic and therefore they should be avoided.

Vector

Description

Vector is another name for a one-dimensional array. It is used in particular for the arrays with elements of logical types: bit and STD_LOGIC (BIT_VECTOR and STD_LOGIC_VECTOR, respectively).

Examples

```
signal Data_Bus : BIT_VECTOR( 7 downto 0 );
-- ...
Data_Bus <= "11000011";
```

Important Notes

- Vectors of logical values are not directly transferable to integer values. Neither STANDARD package nor std_logic_1164 package define any conversion functions from logical vector to an integer or vice-versa. Most synthesis tools, however, support such functions (although these functions are tool-specific).
- No arithmetic functions are allowed on logical vectors.

VITAL

Formal Definition

VITAL (VHDL Initiative Towards ASIC Libraries) is an initiative, which objective is to accelerate the development of sign-off quality ASIC macro-cell simulation libraries written in VHDL by leveraging existing methodologies of model development.

Complete description: IEEE Standard 1076.4.

Description

VITAL, which is now standardized by IEEE, was created by an industry-based, informal consortium in order to accelerate the availability of ASIC (Application Specific Integrated Circuits) libraries for use in industrial VHDL simulators. As a result of the effort a new modeling specification has been created.

VITAL contains four main elements:

- Model Development Specification document, which defines how ASIC libraries should be specified in VITAL-compliant VHDL in order to be simulated in VHDL simulators.
- VHDL package Vital_Timing, defining standard types and procedures that support development of macro-cell timing models. The package contains routines for delay selections, timing violations checking and reporting and glitch detection.
- VHDL package Vital_Primitives, defining commonly used combinatorial primitives provided both as functions and concurrent procedures and supporting either behavioral or structural modeling styles, e.g. VitalAND, VitalOR, VitalMux4, etc. The procedure versions of the primitives support separate pin-to-pin delay path and VitalGlitchOnEvent glitch detection. Additionally, general purpose Truth Tables and State Tables are specified which are very useful in defining state machines and registers.
- VITAL SDF map - specification that defines the mapping (translation) of Standard Delay Files (SDF), used for support timing parameters of real macro-cells, to VHDL generic values.

Modeling Specification

The modeling specification of VITAL defines several rules for VHDL files to be VITAL-compliant. This covers in particular:

- Naming conventions for timing parameters and internal signals, including prefixes for timing parameters, which must be used in the generics specifications (Example 1);
- How to use the types defined in the Vital_Timing package for specifications of timing parameters;
- Methodology of coding styles;
- Two levels of compliance: level 0 for complex models described at higher level, and level 1, which additionally permits model acceleration.

A VITAL-compliant specification consists of an entity with generics defining the timing parameters of the ports (Example 1) and an architecture that can be written in one of two coding styles: either pin-to-pin delay style or distributed delay style.

Pin-to-pin Delay Modeling Style

An architecture that follows this style contains two main parts (Example 2):

- A block called Wire_Delay, which defines input path delays between input ports and internal signals. This block calls the concurrent procedure VitalPropagateWireDelay.
- A process called VitalBehavior. This process may contain declarations of local aliases, constants and variables. It has a very rigid structure and is divided into three parts:
    - ♦ Timing Checks - does calls to procedure VitalTimingCheck which is defined in the package Vital_Timing.
    - ♦ Functionality - makes one or more calls to subprograms contained in the Vital_Primitives package and assignments to internal temporal variables. No wait statements, signal assignments or control structures are allowed here.
    - ♦ Path Delay - contains a call to VitalPropagateDelay for each output signal.

Distributed Delay Modeling Style

In this style the specification (ASIC cell) is composed of structural portions (VITAL primitives), each of which has its own delay. The output is an artifact of the structure, events and actual delays. All the functionality is contained in one block, called Vital_Netlist and this block may contain only calls to primitives defined in the Vital_Primitives package.

Examples

Example 1

```
library IEEE;
use IEEE.std_logic_1164.all;
library VITAL;
use VITAL.Vital_Timing.all;
use VITAL.Vital_Timing;
use VITAL.Vital_Primitives.all;
use VITAL.Vital_Primitives;
entity Counter is
  generic(
    tpd_ClkOut1 : DelayType01 := ( 10 ns, 10 ns );
    ...
  );
  port(
    Reset : in STD_LOGIC := 'U';
    Clk : in STD_LOGIC := 'U';
    Count_Out : out STD_LOGIC_VECTOR( 3 downto 0 )
  );
end entity;
```

This entity is a part of a VITAL-compliant specification of a four-bit synchronous counter with reset. Note that two libraries and three packages are used. In particular, multiple value logic types, defined in std_logic_1164, are standard logical types for VITAL.

The example given here specifies the propagation delay between the `Clk` input and the output number 1. The VITAL prefix tpd determines the timing parameter. The type used is specified in the Vital_Timing package.

Example 2

```
architecture Pin_To_Pin of Counter is
  -- declarations of internal signals
begin
  -- Input path delay
  Wire_Delay: block
  begin
    -- calls to the VitalPropagateWireDelay procedure
    Vital_Timing.VitalPropagateWireDelay ( ... );
    -- ...
  end block;

  -- Behavior section
  VitalBehavior: process( ... )
    -- declarations
  begin
    -- Timing Check
    Vital_Timing.VitalTimingCheck ( ... );
    -- Functionality
    Vital_Primitives.VitalStateTable ( ... );
    -- Path Delay
    Vital_Timing.VitalPropagatePathDelay ( ... );
    Vital_Timing.VitalPropagatePathDelay ( ... );
  end process;
end architecture;
```

The above listed architecture `Pin_To_Pin` is a template of a pin-to-pin modeling style. All its procedure calls should be specified with appropriate parameters.

Example 3

```
architecture Distributed_Delay of Counter is
  -- internal signals declarations
begin
  -- Input path delay
  Vital_Netlist: block
    -- internal declarations of the block
  begin
    -- calls to VITAL primitives, for example
    Vital_Primitives.VitalAND2( ... );
    Vital_Primitives.VitalBuf( ... );
    Vital_Primitives.VitalStateTable( ... );
  end block;
end architecture;
```

The above listed architecture `Distributed_Delay` is a template of a distributed delay modeling style. All its procedure calls should be specified with appropriate parameters.

Wait Statement

Definition

The wait statement is a statement that causes suspension of a process or a procedure.

Complete description: Language Reference Manual IEEE 1076-1993 section § 8.1.

Simplified Syntax

```
wait;
wait on signal_list;
wait until condition;
wait for time;
```

Description

The wait statement suspends the execution of the process or procedure in which it is specified. Resuming the process or procedure depends on meeting the condition(s) specified in the wait statement. There are three types of conditions supported with wait statements: sensitivity clause, condition clause, and timeout clause.

The most often used is the sensitivity clause. A sensitivity list defines a set of signals to which the process is sensitive and causes the process to resume (Example 1).

If a wait statement does not contain a sensitivity list, then an implicit sensitivity list is assumed, one which contains all the signals that are present in that condition. If a process is resumed but no condition is met, then the process will not execute any other statements (Example 2).

The second type of a condition supported with the wait statement is the condition clause. A process is resumed when the logical condition turns true due to a change of any signal listed in the condition (Example 2).

The timeout clause defines the maximum time interval during which the process is not active. When the time elapses, the process is automatically resumed (Example 3).

A single wait statement can have several different conditions. In such a case the process will be resumed when all the conditions are met (Example 4).

If a wait on sensitivity_list is the only wait in the process and the last statement of the process, then it can be substituted by a sensitivity list of a process. See sensitivity list for details.

The syntax of the wait statement allows to use it without any conditions. Such a statement is equivalent to wait until true, which suspends a process forever and will never resume. While in simulation of normal models this is a disadvantage, this particular feature of a wait statement is widely used in testbenches. Example 5 shows an example of a testbench section.

Examples

Example 1

```
signal S1, S2 : STD_LOGIC;
-- ...
process
begin
  -- ...
  wait on S1, S2;
end process;
```

After executing all statements, the process will be suspended on the wait statement and will be resumed when one of the `S1` or `S2` signals changes its value.

Example 2

```
wait until Enable = '1';
-- this is equivalent to
-- loop
-- wait on Enable;
-- exit when Enable= '1';
-- end loop;
```

In this example, the wait statement will resume the process when the `Enable` signal changes its value to '1'. This is equivalent to the loop described in the comment below the first line. Please note that the process is resumed on any change of the `Enable` signal. However, it will awake the rest of the process only when the new value is '1'.

Example 3

```
wait for 50 ns;
```

A process containing this statement will be suspended for 50 ns.

Example 4

```
BIN_COMP : process
begin
  wait on A, B until CLK = '1';
  -- ...
end process;
```

The process `BIN_COMP` is resumed after a change on either `A` or `B` signal, but only when the value of the signal `CLK` is equal to '1'.

Example 5

```
G: process
begin
  G0 <= '1' after 5 ns,
        '0' after 10 ns,
        '1' after 15 ns,
        '0' after 20 ns;
  G1 <= '1' after 5 ns,
        '0' after 15 ns;
  wait;
end process;
```

In this process the values of signals `G1` and `G0` are set to '11', '10', '01', and '00' at the time intervals 5, 10, 15 and 20 ns, respectively. When the `wait` statement is encountered, the process is suspended forever.

Important Notes

- The wait statement can be located anywhere between begin and end process.
- A process with a sensitivity list may not contain any wait statements.

Waveform

Definition

A series of transactions, each of which represents a future value of the driver of a signal. The transactions in a waveform are ordered with respect to time, so that one transaction appears before another if the first represents a value that will occur sooner than the value represented by the other.

Complete description: Language Reference Manual IEEE 1076-1993 § 8.4.

Syntax

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

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

Description

The waveform statement appears on the right-hand side of the signal assignment statement. It supports new values for the signal, possibly together with expected delays when the changes of the values will take effect. Alternatively, in concurrent signal assignments, the reserved word `unaffected` can be used instead of new values.

Each waveform element is composed of an expression whose value will be assigned to the signal driver and optional time expression is defined following the reserved word `after`. The type of the expression appearing in the waveform element must be the same as the type of the target signal (Example 1).

If there is no time expression specified in the waveform element of the signal assignment statement, then the time delay is implicitly declared as after 0 ns (Example 2).

An expression in a waveform can be substituted by the value null. This value can be assigned only to signals declared as guarded, additionally with a resolution function (Example 3).

If the reserved word `unaffected` is used as a waveform in a concurrent signal assignment, it is equivalent to a process executing a null statement.

Evaluation of a waveform element produces a single transaction, i.e. a pair: new value for a signal and time when this value will be assigned. The time is determined by the current simulation time added to the value of time expression in the waveform element.

Examples

Example 1

```
signal D_OUT, E_OUT : BIT_VECTOR( 3 downto 0 );
-- ...
D_OUT <= ( others => '0' ) after 2 ns;
E_OUT <= ( others => '0' ) after 2 ns, ( others => '1' ) after 7 ns;
```

In the first signal assignment statement the "0000" value will be assigned to the driver `D_OUT` after 2ns. In the second assignment, first the "0000" value will be assigned to the `E_OUT` after 2ns and then after additional 5 ns the signal will be assigned the "1111" value.

Example 2

```
C_OUT <= 'X';
if C_OUT = 'X' then -- ...
```

Although the signal assignment will take place "after 0 ns", it is not immediate and when the if statement is executed, `C_OUT` will still have its value unchanged. Update of the `C_OUT` signal will take place at the end of simulation cycle, i.e. when the process is suspended.

Example 3

```
architecture test_null of test is

  function res_fun ( res_val: BIT_VECTOR ) return BIT is
  begin
    for i in res_val'RANGE loop
      if res_val(i)= '1' then
        return '1';
      end if;
      return '0';
    end loop;
  end function;

  signal H : res_fun BIT register;

begin

  P1: process
  begin
    A1: H <= '1' after 10 ns, null after 20 ns;
  end process;

  P2: process
  begin
    A2: H <= '1' after 5 ns, '0' after 10 ns;
  end process;
end architecture;
```

When the signal assignment statements A1 and A2 are executed:

- two transactions are placed in driver of the signal `H` in process `P1` ('1' after 10 ns), (null after 20 ns) and
- two transactions are placed in driver of the signal `H` in process `P2` ('1' after 5 ns), ('0' after 10 ns).

This means that the value of the signal `H` at 5 ns is determined by the resolution function with the value from process `P2` equal to '1' and the value from process `P1` equal to '0'. After 10 ns the resolution function will be called with '1' from process `P1` and '0' from process `P2`. After 20 ns the resolution function will be called with a vector containing only one element ('0', contributed by process `P2`) as the driver from the process `P1` is disconnected.

Important Notes

- The delay values supported with the `after` clause do not accumulate, but all relate to the same simulation time (compare Example 1).

VHDL 2002

Protected Type

Formal Definition

A protected type definition defines a protected type. A protected type implements instantiatiable regions of sequential statements, each of which are guaranteed exclusive access to shared data. Shared data is a set of variable objects that may be potentially accessed as a unit by multiple processes.

Complete description: Language Reference Manual, IEEE Std 1076-2002, § 3.5.

Syntax

```
protected_type_definition ::= protected_type_declaration
  | protected_type_body
```

Description

Protected types encapsulate data and methods that operate on that data, quite similar to classes in object oriented languages.

A protected type can comprise data structures of arbitrary complexity. The data structures are not exposed directly to the user of the type. Instead, a protected type may provide subprograms to read and modify its data. Using subprograms for data access provides several benefits, for example:

- The internal implementation of the type may change without affecting code that uses the type.
- Subprograms can check data integrity, for example a subprogram to set variable RST may report a warning and return prematurely if it is invoked when some other variable (e.g. a variable designating the number of elements in a buffer) is greater than 0.

The definition of a protected type consists of two elements: the protected type declaration and the protected type body. If the protected type is defined in a package, then the protected type declaration should be placed in the package declaration and the protected type body in matching package body.

Objects of protected type are always variables, typically shared variables. Shared variables can be accessed by multiple processes.

Examples

Example 1

The declaration of type Counter provides four procedures (CountUp, CountDown, Reset, and Load) that control the counter and a function (Value) to read back the counter value:

```
type Counter is protected
  procedure CountUp;
  procedure CountDown;
  procedure Reset;
  procedure Load ( N: INTEGER );
  impure function Value return INTEGER;
end protected Counter;
```

The actual implementation of the subprograms is provided in the body of the protected type:

```
type Counter is protected body
  variable cnt: INTEGER := 0;

  procedure CountUp is
  begin
    cnt := cnt + 1;
  end procedure;

  procedure CountDown is
  begin
    cnt := cnt - 1;
  end procedure CountDown;

  procedure Reset is
  begin
    cnt := 0;
  end procedure Reset;

  procedure Load ( N: INTEGER ) is
  begin
    cnt := N;
  end procedure;

  impure function Value return INTEGER is
  begin
    return cnt;
  end function;

end protected body;
```

All subprogram bodies appearing in the body of the Counter type are conformant with subprogram declarations in the declaration of the Counter type. Therefore, they are accessible from outside the protected type body.

Example 2

```
package p is

  type pos_t is record
    x : INTEGER;
    y : INTEGER;
  end record;

  type p_t is protected
    procedure set_pos( p : in pos_t );
    impure function get_pos return pos_t;
  end protected;
end package;

package body p is

  type p_t is protected body
    variable pos : pos_t;

    procedure set_pos( p : in pos_t ) is
    begin
      pos := p;
    end procedure;

    impure function get_pos return pos_t is
    begin
      return pos;
    end function;
  end protected body;

end package body;
```

This example shows a protected type p_t defined in package p. The protected type declaration is located in the package declaration and the protected type body in the package body. The pos_t record type used for the protected type method formal arguments must be available outside the protected type, therefore its definition is provided in the package declaration.

Protected Type Declaration

Formal Definition

A protected type declaration declares the external interface to a protected type.

Complete description: Language Reference Manual, IEEE Std 1076-2002, § 3.5.1.

Syntax

```
protected_type_declaration ::=
  protected
  protected_type_declarative_part
  end protected [ protected_type_simple_name ]

protected_type_declarative_part ::=
  { protected_type_declarative_item }

protected_type_declarative_item ::=
  subprogram_declaration
  | attribute_specification
  | use_clause
```

The protected_type_simple_name item is optional. When present, it must be the name of the type for which the protected type definition is provided.

Description

A protected type declaration provides an interface to the protected type, much like a package declaration provides an interface to the package. Subprograms provided in the protected type declaration define operations on objects of the protected type. Such operations are referred to as methods. Note that object declarations (signals, variables, etc.) are not allowed inside protected type declarations.

The formal parameters of subprograms listed in the protected type declaration must not be of an access type or a file type. They also cannot have elements of either an access type or a file type. The same principle applies to the return type of a function subprogram (i.e. an access type and a file type are not allowed as a function return value).

Examples

Example 1

```
type sparse_arr_t is protected
  procedure init;
  procedure store( address, value : in INTEGER );
  impure function recall( address : INTEGER ) return INTEGER;
end protected;
```

The sparse_arr_t provides three methods. Those methods are available to the user of the protected type. The data that the methods operate on is not directly accessible from outside the protected type body and can be manipulated only through the interface provided in the protected type declaration. As long as the interface stays the same, changes to the internals of the protected type do not affect the syntax correctness of the code using the protected type. Note that the function recall is declared as impure. This is required to allow the function to access data declared inside the protected type body.

Example 2

```
type pr_t is protected
  procedure wr( pos : in INTEGER );
  impure function rd return INTEGER;
end protected;
```

A protected type with just two methods. Note that the name of the type (`pr_t`) can be omitted at the end of the declaration, after the `end protected` keywords.

Protected Type Body

Formal Definition

A protected type body provides the implementation for a protected type.

Complete description: Language Reference Manual, IEEE Std 1076-2002, § 3.5.2.

Syntax

```
protected_type_body ::=
  protected body
  protected_type_body_declarative_part
  end protected body [ protected_type_simple name ]

protected_type_body_declarative_part ::=
  { protected_type_body_declarative_item }

protected_type_body_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
```

The protected_type_simple_name item is optional. When present, it must be the name of the type for which the protected type definition is provided.

Description

The protected type body fully defines internal data structures and subprograms operating on them. All elements of protected type body that are not listed in the protected type declaration are not accessible to the user of the protected type. This implies that all data structures and all non-method subprograms are private by definition.

If the protected type is defined in the declarative part of the architecture, then both the protected type declaration and its body should be placed together in the same region. If the protected type is defined in a package, then the protected type declaration should be placed in the package declaration and the protected type body in the matching package body.

All subprograms defined in the protected type body have full access to internal data structures. This implies that all functions in a protected type that access internal data must be defined as impure functions.

Examples

Example 1

```
type p_t is protected
  procedure set_pos( p : in pos_t );
  impure function get_pos return pos_t;
end protected;

type p_t is protected body
  type acc_type_t is ( acc_read, acc_write );
  type data_acc_t is record
    a_time : TIME;
    a_type : acc_type_t;
  end record;

  variable acc : data_acc_t;
  variable pos : pos_t;

  procedure check_access( a : acc_type_t ) is
  begin
    assert acc.a_time /= now or ( acc.a_type = acc_read and a = acc_read )
      report "Race condition detected"
        severity WARNING;
    acc.a_time := now;
    acc.a_type := a;
  end procedure;

  procedure set_pos( p : in pos_t ) is
  begin
    check_access( acc_write );
    pos := p;
  end procedure;

  impure function get_pos return pos_t is
  begin
    check_access( acc_read );
    return pos;
  end function;

end protected body;
```

The listing shows the protected type body preceded by the protected type declaration.

The body provides an implementation for the two methods (get_pos and set_pos) specified in the declaration and another subprogram (check_access) that cannot be called from outside of the protected type body. Type definitions and variable declarations placed in the protected type body are also available only within the protected type body.

Shared Variables

Formal Definition

A variable declaration that includes the reserved word `shared` is a shared variable declaration. Shared variables are a subclass of the variable class of objects.

Complete description: Language Reference Manual, IEEE Std 1076-2002, § 4.3.1.3.

Syntax

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

Description

A variable must be shared if it is declared:

- Immediately within entity declaration, architecture body, or a block.
- Immediately within a package and the package is not declared within a subprogram, process, or protected type body.

Variables declared immediately within subprograms and processes must not be shared variables. Variables declared immediately within a package must not be shared variables if the package is declared within a subprogram, process, or protected type body.

Shared variables can be accessed by many process instances. VHDL 1993 does not guarantee exclusive access to shared variables so two or more processes running as separate OS threads and writing to the same shared variable could, hypothetically, clobber the data.

VHDL 2002 tackles the problem with protected types. Protected types guarantee exclusive access to shared data, however, do not preclude race conditions. A race occurs if multiple processes access shared data in the same simulation cycle and at least one process modifies the data. A race condition may indicate a flaw in the description and poor coding practices.

Protected type variables cannot be used in expressions and cannot be a target of the variable assignment. Instead, a selected name should be used where the prefix is the variable name and the suffix is the method name.

- NOTE: IEEE Std 1076-2002 requires that all shared variables should be of a protected type. This is an important change compared to IEEE Std 1076-1993, where shared variables could be of any type except for the file type. Aldec's compiler run in VHDL 2002 allows shared variables that are not of a protected type but emits a warning whenever such a variable is used.

Examples

Example 1

```
architecture race of e is

  type p_t is protected
    impure function get_flag return BOOLEAN;
    procedure set_flag;
    procedure reset_flag;
  end protected;

  type p_t is protected body
    variable flag : BOOLEAN;
    impure function get_flag return BOOLEAN is
    begin
      return flag;
    end function;

    procedure set_flag is
    begin
      flag := TRUE;
    end procedure;

    procedure reset_flag is
    begin
      flag := FALSE;
    end procedure;
  end protected body;

shared variable f : p_t;

begin

  p0: process is
  begin
    f.set_flag;
    wait;
  end process;

  p1: process is
  begin
    report BOOLEAN'image( f.get_flag );
    wait;
  end process;

  p2: process is
  begin
    f.reset_flag;
    wait;
  end process;

end architecture;
```

The example shows architecture race that demonstrates a race condition. The declarative part contains shared variable f of the protected type p_t. The type (also declared in the architecture) provides three methods: set_flag, reset_flag, and get_flag. Methods set_flag and reset_flag modify the shared data (i.e. variable flag inside the protected type body). Method get_flag retrieves the shared data.

The architecture contains three processes labeled: p0, p1, and p2. Each process uses one the methods available in the protected type. All processes are activated in the same simulation cycle. The language does not guarantee any particular process execution order, therefore neither the value retrieved by the get_flag method (and printed by the report statement) nor the value of the flag variable at the end of the simulation cycle can be predicted.

Port Map Aspect

Formal Definition

A port map aspect associates signals or values with the formal ports of a block.

Complete description: Language Reference Manual, IEEE Std 1076-2002, § 5.2.1.2.

Syntax

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

Description

VHDL 2002 changes the restrictions related to actuals that can be associated with a formal port in a port map aspect. Those restrictions apply to actuals that are themselves ports.

In VHDL 2002 ports of mode buffer can be used as actuals for formal ports of mode in and inout. Similarly, ports of out and inout are allowed for formal ports of mode buffer. The differences are summarized in the table below.

Formal port of mode	Actuals allowed in VHDL 1993	Actuals allowed in VHDL 2002
out	port of mode out or inout	port of mode out, inout, or buffer
inout	port of mode inout	port of mode inout, or buffer
buffer	port of mode buffer	port of mode out, inout, or buffer

The restrictions are listed in IEEE Std 1076-2002, § 1.1.1.2.

Example

```
library IEEE;
use IEEE.std_logic_1164.all;

entity INNER is
  port(
    PB : buffer STD_LOGIC
  );
end entity;

architecture A of INNER is
begin
end architecture;

library IEEE;
use IEEE.std_logic_1164.all;

entity OUTER is
  port(
    PIO : inout STD_LOGIC
  );
end entity;

architecture A of OUTER is
  component INNER is
    port(
      PB : buffer STD_LOGIC
    );
  end component;
begin
  C1 : INNER port map (
    PB => PIO
  );
end architecture;
```

The listing shows code that is valid in VHDL 2002 and not allowed in VHDL 1993. The port map aspect maps the formal port PB to actual PIO, that is itself a port. Port PB is of mode buffer whereas port PIO is an inout. Such a connection can be used in VHDL 2002 but is flagged as an error in VHDL 1993.

VHDL 2008

Arrays

Predefined Array Types

Formal Definition

The predefined array types are `STRING`, `BOOLEAN_VECTOR`, `BIT_VECTOR`, `INTEGER_VECTOR`, `REAL_VECTOR`, and `TIME_VECTOR`, defined in package `STANDARD`.

Complete description: Language Reference Manual IEEE 1076-2008, § 5.3.2.3.

Syntax

```
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;
type STRING is array ( POSITIVE range <> ) of CHARACTER;

subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;
type BOOLEAN_VECTOR is array ( NATURAL range <> ) of BOOLEAN;
type BIT_VECTOR is array ( NATURAL range <> ) of BIT;
type INTEGER_VECTOR is array ( NATURAL range <> ) of INTEGER;
type REAL_VECTOR is array ( NATURAL range <> ) of REAL;
type TIME_VECTOR is array ( NATURAL range <> ) of TIME;
```

Description

VHDL 2008 defines four new array types: `BOOLEAN_VECTOR`, `INTEGER_VECTOR`, `REAL_VECTOR`, and `TIME_VECTOR`. The values of the types are one-dimensional arrays of the predefined types `BOOLEAN`, `INTEGER`, `REAL`, and `TIME`, indexed by values of the predefined subtype `NATURAL`. The types are defined in package STANDARD.

Types `BIT_VECTOR` and `STRING` were already available in VHDL 1993 and are listed here for completeness.

Examples

Example 1

```
signal flags : BOOLEAN_VECTOR( 0 to 7 ) := ( others => FALSE );
```

Vector `flags` is declared as a `BOOLEAN_VECTOR` with the ascending range 0 to 7. The initial value is assigned to all vector fields using an aggregate with choice `OTHERS`.

Example 2

```
process is begin
signal dta : INTEGER_VECTOR( 3 downto 0 );
signal dt2 : INTEGER_VECTOR( 1 downto 0 ) := ( 5, 6 );

-- ...

dta <= dt2 & dt2;
dta <= ( 0, 0, 1, 0 );
dta(1) <= 3;
dta( 0 to 1 ) <= ( 4, 5 );
dta( 2 to 3 ) <= ( 128, INTEGER'high );
wait;
end process;
```

Two integer vectors are declared. The initial value is assigned to vector `dt2` at declaration. Several possible assignments to vector `dta`, its slices, and elements are shown below the declaration.

Functions Predefined for Array Types

Several operations for array types are declared implicitly. (They do not have to be declared explicitly in VHDL; implicit declarations follow immediately after the type definition.)

Complete description: Language Reference Manual IEEE 1076-2008, § 5.3.2.4.

Functions Minimum and Maximum

Definition and Syntax

Given a type declaration that declares a discrete array type `T`, the following operations are implicitly declared immediately following the type declaration:

```
function MINIMUM ( L, R: T ) return T;
function MAXIMUM ( L, R: T ) return T;
```

In addition, given a type declaration that declares a one-dimensional array type `T` whose elements are of a scalar type `E`, the following operations are implicitly declared immediately following the type declaration:

```
function MINIMUM ( L: T ) return E;
function MAXIMUM ( L: T ) return E;
```

Return Value

If both L and R arguments are specified then

- The `MINIMUM` function returns the value of `L` if `L < R`, or the value of `R` otherwise.
- The `MAXIMUM` function returns the value of `R` if `L < R`, or the value of `L` otherwise.

The comparison is performed using the predefined relational operator for the type.

If only one argument is specified then

- The `MINIMUM` function returns a value that is the least of the elements of `L`. If `L` is null, then the E'HIGH is returned.
- The `MAXIMUM` function returns a value that is the greatest of the elements of `L`. If `L` is null, then the `E'LOW` is returned.

Example 1

```
process
  variable v1, v2 : INTEGER_VECTOR( 1 to 3 );
begin
  v1 := read_channel(1);
  v2 := read_channel(2);
  v1 := minimum( v1, v2 );
  -- ...
  wait;
end process;
```

The example defines two variables (`v1` and `v2`) of type `INTEGER_VECTOR`. Type `INTEGER_VECTOR` is an array of elements of a discrete type so function minimum is available for that type. The function is used to assign the lesser of the two values to variable `v1`.

Example 2

```
  signal buf_pos_arr : INTEGER_VECTOR( 1 to 4 );

begin
  p1: process ( buf_pos_arr )
  begin
    report INTEGER'image ( maximum ( buf_pos_arr ) );
  end process;
```

The process in the listing is sensitive to changes on a vector signal. Whenever the signal changes value, the largest value in the vector is determined using function `MAXIMUM` and reported to the standard output.

Example 3

```
subtype str_t is STRING( 1 to numchars );
type arr_t is array ( NATURAL range <> ) of str_t;

function string_max ( s: in arr_t ) return str_t is
  variable result : str_t;
begin
  if ( s'length > 0 ) then
    result := s( s'left );
    for i in s'left + 1 to s'right loop
      result := maximum ( s(i), result );
    end loop;
  else
    result := ( others => CHARACTER'low );
  end if;
  return result;
end;
```

Function `string_max` shown in the listing scans the array of strings passed as formal parameters and returns the lexically largest string. If the array range is null, then a string consisting of the lower bound of the character type is returned. Strings are compared in pairs. The largest string in each pair is selected with the `MAXIMUM` function.

Note that you cannot use the `MAXIMUM` function directly on the array of string. The language predefines the `MAXIMUM` function for arrays of scalar types but not for arrays of composite types such as an array of strings.

Function To_String

Definition and Syntax

Given a type declaration that declares a one-dimensional array type `T` whose element type is a character type that contains only character literals, the following operation is implicitly declared immediately following the type declaration:

```
function TO_STRING( VALUE: T ) return STRING;
```

Return Value

The `TO_STRING` function returns the string representation of the value of its actual parameter.

Important Notes

- Function `TO_STRING` is not available for arrays of characters because the character type contains elements that are not character literals (`NUL`, `SOH`, `STX`, ...).

Example

```
process is
  type enum_t is ( 'A', 'B', 'C', '1', '2', '3' );
  type arr_t is array ( 0 to 2 ) of enum_t;
  variable arr : arr_t;
begin
  report "Execution started; arr = " & to_STRING( arr );
  -- ...
  wait;
end process;
```

When simulation is initialized, the process in the listing will print

```
Execution started; arr = AAA
```

Additional Functions for the BIT_VECTOR Type

Definition and Syntax

The following functions are implicitly defined for the `BIT_VECTOR` type:

```
function TO_OSTRING( VALUE: BIT_VECTOR ) return STRING;
function TO_HSTRING( VALUE: BIT_VECTOR ) return STRING;
```

Additionally, the following aliases are available:

```
alias TO_BSTRING is TO_STRING [BIT_VECTOR return STRING];
alias TO_BINARY_STRING is TO_STRING [BIT_VECTOR return STRING];
alias TO_OCTAL_STRING is TO_OSTRING [BIT_VECTOR return STRING];
alias TO_HEX_STRING is TO_HSTRING [BIT_VECTOR return STRING];
```

Return Value

Function `TO_OSTRING` returns a string with the octal representation of the value. If the length of the parameter value is not a multiple of three, then '0' elements are implicitly concatenated on the left.

Function `TO_HSTRING` returns a hexadecimal representation of the value. Digits `A-F`, if any, are in the upper case. If the length of the parameter value is not a multiple of four, then '0' elements are implicitly concatenated on the left.

Example

```
type mnemonics_t is ( ADD, JE, JMP, JNE, MOV, POP, PUSH, RET );
type opcodes_t is array ( mnemonics_t range <> ) of BIT_VECTOR( 2 downto 0 );

constant opcodes : opcodes_t :=
(
  "000", "001", "010", "011",
  "100", "101", "110", "111"
);

procedure dump_opcodes is
begin
  for m in mnemonics_t'low to mnemonics_t'high loop
    report TO_HSTRING( opcodes(m) )
      & " | " & TO_BSTRING( opcodes(m) )
      & " | " & TO_STRING(m);
  end loop;
end;
```

The `DUMP_OPCODES` procedure shown in the example prints a table with operation codes in the hexadecimal and binary form followed by their mnemonics.

0	000	ADD
1	001	JE

2	010	JMP
3	011	JNE
4	100	MOV
5	101	POP
6	110	PUSH
7	111	RET

Note that the `TO_STRING` function in the example is used both on arrays (an element of the OPCODES array is itself an array) and on scalars (elements of the `MNEMONICS_T` type).

Attributes (predefined)

Syntax

```
O'subtype
A'element
```

Description

VHDL 2008 adds support for two new attributes, `O'subtype` and `A'element`.

The prefix of attribute `subtype` must be appropriate for an object or its alias. The attribute value is the fully constrained subtype that is the subtype of `O`. For `O` that is an array, the attribute will contain constraints defining index ranges.

The usage of attribute `subtype` is demonstrated in Example 1.

The prefix of attribute `element` must be appropriate for an array object, its alias, or denote an array subtype. The attribute value depending on the type of prefix is:

- The fully constrained element subtype that is the element subtype of A (for A that is an array).
- The element subtype of A (for A that is an array subtype).

Example 2 shows attribute `element` with an array object prefix, in Example 3 an array subtype prefix is used.

Examples

Example 1

```
entity ex is
  port(
    p : in integer
  );
end entity ex;

architecture ex of ex is
  signal s : p'subtype;
begin
end architecture ex;
```

Example 2

```
package p is
  type arr_t is array( 1 to 10) of time;
  signal s : arr_t'element;
end package;
```

Example 3

```
package p is
  type arr_t is array( 1 to 10) of time;
  signal arr: arr_t;
  signal s : arr'element;
end package;
```

Important Notes

For other supported attributes, see the Attributes in the VHDL 1993 section.

Delimited Comment

Formal Definition

A delimited comment starts with a slash immediately followed by an asterisk (`/*`) and extends up to the first subsequent occurrence of an asterisk immediately followed by a slash (`*/`).

Complete description: Language Reference Manual IEEE 1076-2008, § 15.9.

Read Single-line Comment topic for more details about comments that can appear in a VHDL description.

Syntax

```
delimited_comment ::= /* delimited_comment_text */
```

Description

A delimited comment can be started in any line at any position in the line and finished in the same line or any of the subsequent lines of VHDL source code. It is illegal to use the starting comment delimiter (`/*`) without the matching delimiter (`*/`). Comments cannot be nested.

Delimited comments are a convenient way to comment out larger blocks of text. Rather than comment out each single line with two adjacent hyphens, you can enclose a multi line block with just two delimiters (`/*` and `*/`).

Examples

Example 1

```
/*
  Copyright (c) Aldec, Inc.
  All rights reserved.
  Last modified: Date: 2008-07-21 14:29:30
  Revision: 43348
*/
```

Context Clause

Formal Definition

A context clause defines the initial name environment in which a design unit is analyzed.

Complete description: Language Reference Manual IEEE 1076-2008, § 13.4.

Simplified Syntax

```
library lib_name;
use lib_name.package_name.item;
use lib_name.package_name.all;
context lib_name.context_name;
```

Description

The term context clause comprises a library clause, a use clause, and a context reference. Library and use clauses are available in all revisions of the VHDL standard (see Use Clause in the VHDL 1993 section). A context reference is a new addition to the VHDL 2008 standard. A context reference makes all library and use clauses inside the context declaration visible to the compiled design unit. The context declaration must be compiled to a library before a context reference can be used.

See Context Declaration for usage examples.

Context Declaration

Formal Definition

A context declaration defines context items that may be referenced by design units.

Complete description: Language Reference Manual IEEE 1076-2008, § 13.3.

Simplified Syntax

```
context identifier is
  context_clauses;
end [context] [identifier];
```

Description

A context declaration is a convenient way to group several `library` and `use` clauses. The context declaration must be compiled to a library, like any other design unit and then a context reference can be used instead of library and use clauses grouped inside the context declaration.

The library clause in a context declaration cannot define library name WORK and the use clause cannot use a selected name with library name WORK as prefix.

Examples

```
context tb_env is
  library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.std_logic_arith.all;
  use IEEE.std_logic_unsigned.all;
  use IEEE.std_logic_textio.all;
end context;
```

The example shows a context that groups several packages from the IEEE library. The context reference can be then used as follows:

```
context work.tb_env;

entity e is port(
  p : out STD_LOGIC
);
end entity e;
```

Note that `STD_LOGIC` type can be used in the entity declaration because the `std_logic_1164` package where the `STD_LOGIC` type is defined is visible through the context reference.

Matching Selected Signal Assignment

Formal Definition

A matching selected signal assignment statement is equivalent to the sequential case statement with the question mark delimiters following both occurrences of the reserved word `case`.

Complete description: Language Reference Manual IEEE 1076-2008, § 10.5.4.

Simplified Syntax

```
with expression select?
  target <= [ delay_mechanism ] selected_waveforms;
```

Description

VHDL 2008 distinguishes two types of a selected signal assignment statement: an ordinary selected signal assignment statement and a matching selected signal assignment statement. For a description of an ordinary selected signal assignment statement, see Selected Signal Assignment in the VHDL 1993 section. The matching selected signal assignment statement is indicated by the question mark after the `select` keyword. When the question mark modifier is used, the expression value is compared to the choice using the matching equality operator (?=) (see Matching Relational Operators). Additionally, a don't care value in the expression is not allowed and triggers a runtime error. For operands of type `STD_LOGIC`, the matching equality operator, unlike the ordinary equality operator (=) returns true when comparing '1' against 'H', '0' against 'L', or '-' against any other value.

Examples

```
with sel select? dout <=
  'a' when "1--",
  'b' when "01-",
  'c' when "001",
  'd' when others;
```

The example shows a priority encoder implemented with the select statement. Let us assume that signal `SEL` is assigned the following waveform at time 0:

```
sel <= "100" after 100 ns,
  "0H0" after 200 ns,
  "H00" after 300 ns,
  "001" after 400 ns,
  "00-" after 500 ns;
```

The following assignments will be made to signal DOUT:

- 'a' at 100 ns ("100" in the expression matches "1--").
- 'b' at 200 ns ("0H0" in the expression matches "01-").
- 'a' at 300 ns ("H00" in the expression matches "1--").
- 'c' at 400 ns ("001" in the expression matches "1--").

Finally, at time 500 ns the simulator will report a fatal error and terminate the simulation because the select expression (SEL) contains a don't care value.

Important Notes

- VHDL 2008 allows using selected signal assignments not only as concurrent statements but also as sequential statements. See Sequential Signal Assignment for details.

Matching Case Statement

Formal Definition

The matching case statement selects for execution one of several alternative sequences of statements; the alternative is chosen based on the value of the associated expression. For expressions of type `STD_LOGIC`, the don't care ('-') value in a choice matches any value in the expression.

Complete description: Language Reference Manual IEEE 1076-2008, § 10.9.

Simplified Syntax

```
case? expression is
  when choice => sequential_statements
  when choice => sequential_statements
  ...
end case?;

stmnt_label: case? expression is
  when choice => sequential_statements
  when choice => sequential_statements
  ...
end case? stmnt_label;
```

Description

VHDL 2008 distinguishes two types of case statements: an ordinary case statement and a matching case statement. For a description of an ordinary case statement, see Case Statement in the VHDL 1993 section.

The matching case statement is indicated by a question mark after both the opening and the closing `case` keyword. The matching case statement can be used with expressions of type `BIT`, `STD_ULOGIC`, or one dimensional arrays whose elements are of type `BIT` or `STD_ULOGIC`.

For expressions of type `BIT` (and arrays of `BIT`), there is no difference between the matching case statement and an ordinary case statement. For expressions of type `STD_ULOGIC` (and arrays of `STD_ULOGIC`), the following properties distinguish the matching case statement from the ordinary case statement:

- '1' and 'H' match.
- '0' and 'L' match.
- The don't care ('-') value in a choice matches any value in the expression.
- The don't care ('-') value in the expression is not allowed and results in a runtime error.

Examples

Example 1

```
case? s is
  when "1--" => report "select 2";
  when "01-" => report "select 1";
  when "001" => report "select 0";
  when others => report "select none";
end case?;
```

The code in the listing shows a priority encoder. The case expression `S` is of type `STD_LOGIC_VECTOR( 2 downto 0 )`.

When the leftmost element of expression `S` equals '1', the first branch is executed (that is, `select 2` is reported to the standard output), regardless of the values of other elements in the expression. When the middle element equals '1' (and the leftmost element equals '0'), `select 1` is reported. Finally, when the rightmost element equals '1' (and remaining elements equal '0'), `select 0` is reported. When the expression `S` does not match any of the three choices, choice OTHERS is executed.

Operators

Condition Operator

Definition and Syntax

The unary operator ?? is implicitly declared for type `BIT` in package `STD.STANDARD` and for type `STD_ULOGIC` in package `IEEE.std_logic_1164`.

```
function "??" ( anonymous: BIT ) return BOOLEAN;
function "??" ( l : STD_ULOGIC ) return BOOLEAN;
```

Complete description: Language Reference Manual IEEE 1076-2008, § 9.2.9.

Return Value

- When applied to a value of type `BIT`, the operator returns TRUE if the BIT value equals '1'. Otherwise it returns FALSE.
- When applied to a value of type `STD_ULOGIC`, the operator returns TRUE if the `STD_ULOGIC` value equals '1' or 'H'. For other values, it returns FALSE.

Important Notes

The condition operator can be implicitly applied to an expression used as a condition in the following places:

- After keyword `until` in the condition clause of a `wait` statement.
- After `assert` either in the assertion statement or in the concurrent assertion statement.
- After keyword `if` or `elsif` in the if statement.
- After keyword `while` in a while iteration scheme of the loop statement.
- After keyword `when` in the concurrent conditional signal assignment statement.
- After keyword `when` in the next statement.
- After keyword `when` in the exit statement.
- After keyword `if` in an if generate statement.
- The guard expression in a block statement.

Examples

Example 1

```
architecture arch of example is
  signal a_bit: BIT;
  signal b_std: STD_ULOGIC;
  signal c_bool: BOOLEAN;
begin
  test: process ( a_bit, b_std, c_bool )
  begin
```

```
    if ( ?? a_bit ) and ( ?? b_std ) and c_bool then
      -- ...
    else
      -- ...
    end if;
  end process;
end;
```

The example shows the explicit use of the ?? operator. Note that the signals A_BIT, B_STD, and C_BOOL are of different types (BIT, STD_ULOGIC, and BOOLEAN, respectively). The condition operator used on signals of type BIT and STD_ULOGIC returns BOOLEAN values. Three BOOLEAN values are then used as operands of logical AND operators.

Example 2

```
architecture arch of example is
  signal x_std: STD_ULOGIC;
begin
  p1: process
  begin
    -- ...
    wait until x_std;
    -- ...
    if x_std then
      -- ...
    end if;
  end process;
end;
```

The example shows implicit use of the condition operator. The operator is applied twice to the x_std signal: in the wait until statement and in the if statement.

Matching Relational Operators

Matching relational operators are predefined for type BIT and BIT_VECTOR in package STD.STANDARD and for type STD_ULOGIC and STD_ULOGIC vector in package IEEE.std_logic_1164.

The set of matching relational operators includes six operators: ?=, ?/=, ?<, ?<=, ?>, and ?>=. Each *matching* relational operator has a counterpart *ordinary* relational operator. The ordinary relational operators (=, /=, <, <=, >, and >=) are available since VHDL 1993. Matching operators for operands of type BIT and STD_ULOGIC return values of type BIT and STD_ULOGIC, respectively. Note their matching equivalents return BOOLEAN values.

Equality and Inequality Operators

Definition and Syntax

Binary operators that test for equality and inequality of operands.

Complete description: Language Reference Manual IEEE 1076-2008, § 9.2.3.

```
function "?=" ( anonymous, anonymous: BIT ) return BIT;
function "?/=" ( anonymous, anonymous: BIT ) return BIT;
function "?=" ( anonymous, anonymous: BIT_VECTOR ) return BIT;
function "?/=" ( anonymous, anonymous: BIT_VECTOR ) return BIT;

function "?=" ( l, r : STD_ULOGIC ) return STD_ULOGIC;
function "?/=" ( l, r : STD_ULOGIC ) return STD_ULOGIC;

function "?=" ( l, r : STD_ULOGIC_VECTOR ) return STD_ULOGIC;
function "?/=" ( l, r : STD_ULOGIC_VECTOR ) return STD_ULOGIC;
```

Return Value

- Matching equality (?=) and inequality (`?/=`) operators with operands of type `BIT` return '1' where the equivalent ordinary operator returns TRUE. Otherwise '0' is returned.
- For operands of type `STD_ULOGIC`, the matching equality operator returns an `STD_ULOGIC` value according to the table below. Note that `'1'` is returned if either of the operands equals `'-'`. Also note that expressions `'1' = 'H'` and `'0' = 'L'` both return '1'.

?=	**'U'**	**'X'**	**'0'**	**'1'**	**'Z'**	**'W'**	**'L'**	**'H'**	**'-'**
'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'1'
'X'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'1'
'0'	'U'	'X'	'1'	'0'	'X'	'X'	'1'	'0'	'1'
'1'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'1'
'Z'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'1'
'W'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'1'
'L'	'U'	'X'	'1'	'0'	'X'	'X'	'1'	'0'	'1'
'H'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'1'
'-'	'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'

- The matching inequality operator (`?/=`) returns the result of applying the `not` operator to the result of applying the ?= operator to the operands.

Ordering Operators

Definition and Syntax

Binary operators that test ordering of operands.

Complete description: Language Reference Manual IEEE 1076-2008, § 9.2.3.

```
function "?<" ( anonymous, anonymous: BIT ) return BIT;
function "?<" ( l, r : STD_ULOGIC ) return STD_ULOGIC;

function "?<=" ( anonymous, anonymous: BIT ) return BIT;
function "?<=" ( l, r : STD_ULOGIC ) return STD_ULOGIC;
function "?>" ( anonymous, anonymous: BIT ) return BIT;
function "?>" ( l, r : STD_ULOGIC ) return STD_ULOGIC;

function "?>=" ( anonymous, anonymous: BIT ) return BIT;
function "?>=" ( l, r : STD_ULOGIC ) return STD_ULOGIC;
```

Return Value

- Matching ordering operators with operands of type BIT return `'1'` where the equivalent ordinary operator returns TRUE. Otherwise '0' is returned.
- The matching ordering operator ?< for operands of type `STD_ULOGIC` returns an `STD_ULOGIC` value defined in the table below. The header column contains the left argument, the header row, the right argument.

?<	**'U'**	**'X'**	**'0'**	**'1'**	**'Z'**	**'W'**	**'L'**	**'H'**	**'-'**
'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'X'
'X'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'0'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'1'	'U'	'X'	'0'	'0'	'X'	'X'	'0'	'0'	'X'
'Z'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'W'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'L'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'H'	'U'	'X'	'0'	'0'	'X'	'X'	'0'	'0'	'X'
'-'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'

- The value of the remaining ordering operators applied to operands of type `STD_ULOGIC` can be obtained with the help of the `NOT`, `OR`, ?<, and ?= operators.

Matching ordering operator	**Equivalent expression**
a ?<= b	(a ?< b) or (a ?= b)
a ?> b	not ((a ?< b) or (a ?= b))
a ?>= b	not (a ?< b)

Reduction Operators

Definition and Syntax

The term reduction operators refers to unary logical operators, i.e logical operators with one operand. The reduction operators are implicitly declared in the `STD.STANDARD` package for types `BIT_VECTOR` and `BOOLEAN_VECTOR`.

```
function "and" ( anonymous: BIT_VECTOR ) return BIT;
function "or" ( anonymous: BIT_VECTOR ) return BIT;
function "nand" ( anonymous: BIT_VECTOR ) return BIT;
function "nor" ( anonymous: BIT_VECTOR ) return BIT;
function "xor" ( anonymous: BIT_VECTOR ) return BIT;
function "xnor" ( anonymous: BIT_VECTOR ) return BIT;

function "and" ( anonymous: BOOLEAN_VECTOR ) return BOOLEAN;
function "or" ( anonymous: BOOLEAN_VECTOR ) return BOOLEAN;
function "nand" ( anonymous: BOOLEAN_VECTOR ) return BOOLEAN;
function "nor" ( anonymous: BOOLEAN_VECTOR ) return BOOLEAN;
function "xor" ( anonymous: BOOLEAN_VECTOR ) return BOOLEAN;
function "xnor" ( anonymous: BOOLEAN_VECTOR ) return BOOLEAN;
```

Additionally, an explicit declaration is available for the STD_ULOGIC type in package `IEEE.std_logic_1164`.

Complete description: Language Reference Manual IEEE 1076-2008, § 9.2.2.

Return Value

- For `AND`, `OR`, and `XOR`, the result of applying the operator to operand `O` is the same as applying the equivalent binary operator to operands `L` and `R`, where `L` is the leftmost element of `O` and `R` is the result of applying the unary operator to the remaining rightmost elements of `O`. If the operand of the unary `AND` operator is a null array, then the value of the expression is '1' (for BIT arrays) or TRUE (for BOOLEAN arrays). For operators `OR` and `XOR` used with a null array operand the result is '0' (for BIT arrays) or FALSE (for BOOLEAN arrays).
- For `NAND`, `NOR`, and `XNOR`, the result is the negated value returned by operands `AND`, `OR`, and `XOR`, respectively.

Examples

Example 1

```
  signal line_sel : BIT_VECTOR( 7 downto 0 );
  signal active : BIT;
begin
  active <= or line_sel;
  -- ...
```

Signal `ACTIVE` in the listing is driven with the result of the unary `OR` operator on the `LINE_SEL` vector. This is equivalent to the following assignment using the binary `OR` operator (with unary `OR` in the right operand).

```
active <= line_sel(7) or ( or line_sel( 6 downto 0 ) );
```

The result of the unary operator in the right operand can be written as

```
line_sel(6) or ( or line_sel( 5 downto 0 ) );
```

and so on.

Example 2

```
all_off <= nor line_sel;
```

The result of the unary `NOR` operator is assigned to signal `all_off`. This is equivalent to the following assignment:

```
all_off <= not ( or line_sel );
```

Note that the parenthesis cannot be omitted.

Packages

Uninstantiated Packages

Formal Definition

An uninstantiated package is a package whose header contains a generic clause and no generic map aspect.

Complete description: Language Reference Manual IEEE 1076-2008, § 4.7.

Simplified Syntax

```
package_declaration ::=
  package identifier is
    package_header
    package_declarative_part
end [ package ] [ package_simple_name ];

package_header ::=
  generic_clause
```

Description

VHDL 2008 enhances packages by allowing generics in package declarations. The generic list in a package is similar to generic lists in entities or components. A package with a generic list is called an uninstantiated package. Prior to using such a package you must instantiate it, much like instantiating a component. For information on package instantiation, see Package Instantiation Declaration. Note that instantiation is necessary even if default values are provided for all package generics.

Note that VHDL 2008 introduces the term *simple package* to refer to a package without a generics list.

Examples

Example 1

```
library IEEE;
use IEEE.std_logic_1164.all;

package busutils is
    generic (
        DATA_W: NATURAL;
        ADDR_W: NATURAL
    );

    subtype DTA_t is STD_LOGIC_VECTOR( DATA_W - 1 downto 0 );
    subtype ADR_t is STD_LOGIC_VECTOR( ADDR_W - 1 downto 0 );
    signal CTL: STD_LOGIC_VECTOR( 3 downto 0 );
    -- code omitted for brevity
```

```
end package busutils;

package body busutils is
    -- code omitted for brevity
end package body;
```

The example shows package *busutils*. The generics list in the package contains two declarations. The contents of package `busutils` are not accessible because the package is not instantiated.

Example 2

This example shows an illegal usage of package `busutils`. The `use` clause triggers an error, because the package is uninstantiated. For information on package instantiation, see Package Instantiation Declaration.

```
use work.busutils.all; -- ERROR
```

Package Instantiation Declaration

Formal Definition

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

Complete description: Language Reference Manual IEEE 1076-2008, § 4.9.

Simplified Syntax

```
package_instantiation_declaration ::=
  package identifier is new uninstantiated_package_name
  [ generic_map_aspect ];
```

Description

A package with a generic list (i.e. an uninstantiated package, see Uninstantiated Packages) must be instantiated before declarations within that package can be referred to. The instantiation declaration provides:

- An identifier for the uninstantiated package.
- A generic map aspect with generic mappings, much like entity instantiation and component instantiation statements. The generic map aspect can be omitted if default values are provided for all formal generics (i.e. generics in the uninstantiated package).

After the package is instantiated, you can access its contents by using the instance name as a prefix. Likewise, you can use the package instance name in a `use` statement.

Examples

Example 1

```
package bus_d16_a16 is new busutils
generic map (
    DATA_W => 16,
    ADDR_W => 16
);
```

The example shows how package `busutils` is instantiated as package `bus_16d_16a`. (For the declaration of the uninstantiated package, see Example 1 in Uninstantiated Packages.)

Example 2

```
use work.bus_d16_a16.all;
entity e is
  -- Code omitted for brevity
end;
```

The example shows the `use` clause for package `bus_d16_a16`. Note that the package contents can be accessed only via the names of package instance(s), not through the name of the uninstantiated package.

Generic-mapped Packages

Formal Definition

A package with a header containing both a generic clause and a generic map aspect is called a generic-mapped package.

Complete description: Language Reference Manual IEEE 1076-2008, § 4.7.

Simplified Syntax

```
package_declaration ::=
  package identifier is
    package_header
    package_declarative_part
end [ package ] [ package_simple_name ];

package_header ::=
  generic_clause
  generic_map_aspect;
```

Description

A generic-mapped package declares a package containing generics and immediately instantiates that package. Further instantiations of the package are not allowed. The practical use for generic-mapped packages is limited. Normally, you would declare an uninstantiated package (see Uninstantiated Packages) and then instantiate it (see Package Instantiation Declaration).

Examples

Example 1

```
library IEEE;
use IEEE.std_logic_1164.all;

package busutils is
    generic (
        DATA_W: NATURAL;
        ADDR_W: NATURAL
    );

    generic map (
        DATA_W => 16,
        ADDR_W => 16
    );

    subtype DTA_t is STD_LOGIC_VECTOR( DATA_W - 1 downto 0 );
    subtype ADR_t is STD_LOGIC_VECTOR( ADDR_W - 1 downto 0 );
    signal CTL: STD_LOGIC_VECTOR( 3 downto 0 );
end package busutils;

package body busutils is
    -- code omitted for brevity
end package body;
```

The example shows a generic-mapped package, i.e. a package containing both a generic clause and a generic map.

Example 2

```
use work.busutils.all;
entity e is
  -- Code omitted for brevity
end entity e;
```

The package declared in Example 1 is referred to using simple name (`busutils`).

Example 3

```
package bus_16_16 is new busutils -- ERROR
generic map (
    DATA_W => 16,
    ADDR_W => 16
);
```

The example in the listing triggers an error because instantiations of generic-mapped packages are illegal.

Interface Packages

Formal Definition

An interface package declaration declares an interface package that appears as a generic of another package.

Simplified Syntax

```
interface_package_declaration ::=
      package identifier is new uninstantiated_package_name interface_package_generic_map_aspect
interface_package_generic_map_aspect ::=
      generic_map_aspect
      | generic map ( <> )
      | generic map ( default )
```

Description

An interface package provides a means for the environment to determine an instance of an uninstantiated package to be visible in a particular portion of a description by associating an actual instantiated package with the formal interface package.

Examples

Example 1

```
package helper is
  generic (
    i: integer;
    j: integer
  );
end package helper;

package helper_instance is new helper
  generic map(
    i => 1,
    j => 1
  );

package top is
  generic (
    g: integer;
    package iface_pack is new helper generic map ( <> )
  );
end package;

package top_instance is new top
  generic map (
    g => 1,
    iface_pack => helper_instance
  );
```

The example shows a declaration of an interface package `iface_pack`, i.e.

```
package iface_pack is new helper generic map ( <> )
```

The declaration is located inside package `top`, which must itself be instantiated. The declaration specifies the requirements for the interface package, namely:

- The actual package specified at instantiation of package `top` must be an instance of package `helper`.
- The generics for the instance of package `helper` can be arbitrary. This is indicated by the generic map in the form of the box (<>) symbol.

When package `top` is instantiated as `top_instance`, package `helper_instance` (which is an instance of package `helper`) is specified as one of the generics (alongside generic `g`).

Port Map Aspect

Formal Definition

A port map aspect associates signals or values with the formal ports of a block.

Complete description: Language Reference Manual IEEE 1076-2008, § 6.5.7.3, § 14.3.5.

Simplified Syntax

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

Description

The port map aspect is used in component instantiation statements. (See Component Instantiation Component Instantiation> in the VHDL 1993 section.) VHDL 2008 enhances the port map aspect by allowing the use of expressions. An expression can be associated with a formal port of mode IN in the actual part of a given association. The expression specified in the port map aspect can employ predefined operators, i.e. relational, adding, logical, shift, sign, multiplying, or exponentiating operator. The operands of the expression can be either scalars or vectors.

Example 1

```
architecture arch of example is

  component L1_INT is
  port(
    c : in integer
  );
  end component;
  signal a, b : INTEGER;

begin
  UUT: L1_INT
    port map (
      c => a + b
    );
end;
```

In this example, the formal port `C` is associated with an actual represented by an expression containing an addition operator. The operands of the expression are scalars of the `INTEGER` type.

Example 2

```
architecture arch of example is

  component L2_BIT is
  port(
    c : in BIT
  );
  end component;
  signal a, b : BIT;

begin
  UUT: L2_BIT
  port map (
    c => a & b
  );
end;
```

This example uses an expression with the logical `and` operator. The formal port `C` is associated with the actual represented by the expression whose operands are scalars of type `BIT`.

Example 3

```
architecture behavioral of tb is

  component L3_STD is
    port(
      c : in STD_LOGIC_VECTOR( 1 downto 0 )
    );
  end component;
  signal a, b : STD_LOGIC_VECTOR( 3 downto 0 );

begin
  UUT: L3_STD
    port map (
      c => a( 1 downto 0 ) and b( 1 downto 0 )
    );
end;
```

In Example 3, the formal port `C` is associated with the actual represented by an expression containing the logical `AND` operator. The operands of the expression are arrays.

Process Sensitivity List

Formal Definition

A process sensitivity list is either keyword `all` or a sensitivity list that follows the keyword `process`.

Complete description: Language Reference Manual IEEE 1076-2008, § 11.3.

Syntax

```
process_sensitivity_list ::=
  all
| sensitivity_list
```

Description

A process sensitivity list is a part of the process statement. (See Process Statement Process Statement> in the VHDL 1993 section.) VHDL 2008 enhances the process sensitivity list by allowing the use of the reserved word `all`. The reserved word `all` can be used instead of a list of explicitly defined signals. A sensitivity list created with the reserved word `all` includes:

- signals used on the right side of the assignment statements
- signals used for expressions occurring in the index names or slice names in the assignment target
- signals used in condition expressions in the IF statement and CASE statement
- signals used in iteration schemes in loop statements
- signals used in `assert`, `report`, and `exit` statements
- signals used as actual arguments in a procedure call statement and associated with formal parameters of mode IN and INOUT

Additionally, if a process is a parent of a subprogram (i.e. there is a subprogram call in the process), then the subprogram body is also scanned for signals matching the criteria in the list above. This applies only to subprograms declared in the unit containing the process statement. For subprograms declared in other units, the compiler prints a warning that the sensitivity list cannot be inferred.

Examples

Example 1

```
process ( all )
begin
  a <= b;
end process;
```

The process in the listing is sensitive to changes on signal B. (It is not sensitive to changes on A.)

Example 2

```
process ( all )
begin
  report "Process activated";
  report "a'event = " & BOOLEAN'image ( a'event );
  report "b'event = " & BOOLEAN'image ( b'event );
  a <= b;
end process;
```

The process in the listing is sensitive to changes on both A and B. Signal A is on the sensitivity list because it used in the report statement.

Example 3

```
process ( all )
begin
  if a='1' then
    y <= "01";
  elsif b='1' then
    y <= "10";
  else
   y <= "00";
  end if;
end process;
```

The process in the listing is sensitive to changes on signals A and B. Those signals are used in conditions in the IF statement. Signal Y is not on the sensitivity list.

Example 4

```
architecture a of example is
  signal m, n : INTEGER;
  signal s1, s2 : STD_LOGIC_VECTOR( 3 downto 0 );
  signal en : STD_LOGIC;

  procedure check_range is
  begin
    assert m >= n report "Incorrect range.";
  end procedure check_range;

begin
  process ( all )
  begin
    check_range;
    s1 <= s2;
  end process;
  -- ...
end;
```

Process in the listing is sensitive to changes on signal `S2`, `M`, and `N`. Signals `M` and `N` are used by the procedure `CHECK_RANGE` called by the process and declared in the same design unit (i.e. in architecture `A`).

Sequential Signal Assignment

Formal Definition

A sequential signal assignment is a signal assignment inside a subprogram or a process.

Complete description: Language Reference Manual IEEE 1076-2008, § 10.5.

Simplified Syntax

```
signal_name <= [delay_mechanism ] waveform;

signal_name <= [delay_mechanism ] waveform_1 when condition_1 else
  waveform_2 when condition_2 else
  ...
  waveform_n;

with selection select
  signal_name <= [delay_mechanism ] waveform_1 when choice1,
  waveform_2 when choice2,
  ...
  waveform_n when others;

with selection select?
  signal_name <= [delay_mechanism ] waveform_1 when choice1,
  waveform_2 when choice2,
  ...
  waveform_n when others;
```

Description

VHDL 2008 enhances the sequential process assignment statement. The previous revisions of the VHDL standard allowed only simple signal assignments as sequential statements. (See Signal Assignment in VHDL 1993 section.) Beginning with VHDL 2008, a conditional signal assignment and a selected signal assignment (both ordinary and matching) can be used as sequential statements. The conditional signal assignment can be translated in the equivalent if statement; the selected signal assignment into the equivalent case statement.

Examples

Example 1

```
process ( all )
begin
  t <= 1 when c = 0 else
       0 when c = 1 else
      -1;
  -- ...
  -- more sequential statements
  -- ...
end process;
```

The conditional signal assignment shown in the listing is equivalent to the following if statement:

```
process ( all )
begin
  if c = 0 then
    t <= 1;
  elsif c = 1 then
    t <= 0;
  else
    t <= -1;
  end if;
  -- ...
  -- more sequential statements
  -- ...
end process;
```

Example 2

```
process ( all )
begin
  with s select dout <=
    a when "11",
    b when "01",
    c when "10",
    d when others;
  -- ...
  -- more sequential statements
  -- ...
end process;
```

The selected signal assignment shown in the listing is equivalent to the following case statement:

```
process ( all ) is
begin
  case s is
    when "11" => dout <= a;
    when "01" => dout <= b;
    when "10" => dout <= c;
    when others => dout <= d;
  end case;
  -- ...
  -- more sequential statements
  -- ...
end process;
```

Variable Assignment

Formal Definition

A variable assignment statement replaces the current value of a variable with a new value specified by an expression.

Complete description: Language Reference Manual IEEE 1076-2008, § 10.6.1.

Simplified Syntax

```
variable_name := expression;

variable_name := expression_1 when condition_1 else
                 expression_2 when condition_2 else
                 ...
                 expression_n;

with selection select
  variable_name := expression_1 when choice1,
                   expression_2 when choice2,
                   ...
                   expression_n when others;

with selection select?
  variable_name := expression_1 when choice1,
                   expression_2 when choice2,
                   ...
                   waveform_n when others;
```

Description

VHDL 2008 introduces two new types of the variable assignment statement: a conditional variable assignment and a selected variable assignment. Variable assignment statements defined in VHDL 1993 (see Variable Assignment in the VHDL 1993 section) are henceforth referred to as simple variable assignments.

A conditional variable assignment can be translated into the equivalent if statement (see Example 1); a selected variable assignment into the equivalent case statement (see Example 2). The selected assignment can be either ordinary or matching. The matching selected assignment is indicated by the question mark modifier following the `select` keyword. When the question mark modifier is used, the expression value is compared to the choice using the matching equality operator (?=) (see Matching Relational Operators). Additionally, a don't care value in the expression is not allowed and triggers a runtime error. The matching selected assignment has the equivalent matching case statement. (See Matching Case Statement.)

Examples

Example 1

```
v := '0' when s = "00" else
     '1' when s = "01" else
     'Z';
```

The conditional variable assignment in the listing is equivalent to the following if statement:

```
if s = "00" then
  v := '0';
elsif s = "01" then
  v := '1';
else
  v := 'Z';
end if;
```

Example 2

```
with s select v :=
  '0' when "00",
  '1' when "01",
  'Z' when others;
```

The selected variable assignment shown in the listing is equivalent to the following case statement:

```
case s is
  when "00" => v := '0';
  when "01" => v := '1';
  when others => v := 'Z';
end case;
```

Package Reference

STD.STANDARD

Overview

Package STANDARD predefines a number of types, subtypes and functions. An implicit context clause naming this package is assumed to exist at the beginning of each design unit. Package STANDARD must not be modified by the user.

FOREIGN Attribute

The package STANDARD defines the FOREIGN attribute that may be associated only with architectures or with subprograms. In case of subprograms, the FOREIGN attribute specification must appear in the declarative part in which the subprogram is declared. Subprograms decorated with the FOREIGN attribute and both declarative and statement parts of design entities whose architectures are so decorated are subject to special elaboration rules.

Syntax

```
attribute FOREIGN : STRING;
```

Types

BOOLEAN Type

BOOLEAN is an enumerated type consisting of two values: FALSE and TRUE. The BOOLEAN type values are returned by all comparison functions.

Syntax

```
type BOOLEAN is ( FALSE, TRUE );
```

BIT Type

BIT is an enumerated type consisting of two values: â 0â   and â 1â .

Syntax

```
type BIT is ( '0', 1' )
```

CHARACTER Type

CHARACTER is an enumerated type including three groups of ASCII characters:

- standard control characters represented by literal mnemonics
- standard set visible characters represented by characters in apostrophes
- extended set characters, both visible and control, represented by characters in apostrophes or literal mnemonics consisting of the C letter and three decimal digits representing a relevant ASCII code, e.g. C128, C147.

Syntax

```
type CHARACTER is (
  NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
  BS, HT, LF, VT, FF, CR, SO, SI,
  DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
  CAN, EM, SUB, ESC, FSP, GSP, RSP, USP,

  ' ', '!', '"', '#', '$', '%', '&', ''',
  '( ', ' )', '*', '+', ',, '-', '.', '/',
  '0', '1', '2', '3', '4', '5', '6', '7',
  '8', '9', ':', ';', '<', '=', '>', '?',

  '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
  'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
  'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
  'X', 'Y', 'Z', '[', '\', ']', '^', '_',

  '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
  'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
  'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
  'x', 'y', 'z', '{', '|', '}', '~', DEL,
  C128, C129, C130, C131, C132, C133, C134, C135,
  C136, C137, C138, C139, C140, C141, C142, C143,
  C144, C145, C146, C147, C148, C149, C150, C151,
  C152, C153, C154, C155, C156, C157, C158, C159,

  ' ', 'Â¡', 'Â¢', 'Â£', 'Â¤', 'Â¥', 'Â¦', 'Â§',
  'Â¨', 'Â©', 'Âª', 'Â«', 'Â¬', 'Â–', 'Â®', 'Â¯',
  'Â°', 'Â±', 'Â²', 'Â³', 'Â´', 'Âµ', 'Â¶', 'Â·',
  'Â¸', 'Â¹', 'Âº', 'Â»', 'Â¼', 'Â½', 'Â¾', 'Â¿',

  'Ã ', 'Ã ', 'Ã ', 'Ã ', 'Ã ', 'Ã ', 'Ã ', 'Ã ',
  'Ã ', 'Ã ', 'Ã ', 'Ã ', 'Ã ', 'Ã ', 'Ã ', 'Ã ',
  'Ã ', 'Ã ', 'Ã ', 'Ã ', 'Ã ', 'Ã ', 'Ã ', 'Ã ',
  'Ã ', 'Ã ', 'Ã ', 'Ã ', 'Ã ', 'Ã ', 'Ã ', 'Ã ',

  'Ã ', 'Ã¡', 'Ã¢', 'Ã£', 'Ã¤', 'Ã¥', 'Ã¦', 'Ã§',
  'Ã¨', 'Ã©', 'Ãª', 'Ã«', 'Ã¬', 'Ã–', 'Ã®', 'Ã¯',
  'Ã°', 'Ã±', 'Ã²', 'Ã³', 'Ã´', 'Ãµ', 'Ã¶', 'Ã·',
  'Ã¸', 'Ã¹', 'Ãº', 'Ã»', 'Ã¼', 'Ã½', 'Ã¾', 'Ã¿' );
```

SEVERITY_LEVEL Type

SEVERITY_LEVEL is an enumerated type used in `severity` clauses of Assertion Statement. The type consists of four values: NOTE, WARNING, ERROR and FAILURE, representing the severity level of the text string reported to the user by the assertion statement.

Syntax

```
type SEVERITY_LEVEL is ( NOTE, WARNING, ERROR, FAILURE );
```

INTEGER Type

INTEGER is a numeric type representing mathematical integer numbers within the implementation defined range. Aldec's simulator defines the range from -2147483648 to +2147483647. All of the basic mathematical functions such as addition, subtraction, multiplication, and division can be applied to operands of the INTEGER type.

Syntax

```
type INTEGER is range -2147483648 to 2147483647;
```

REAL Type

REAL is a numeric type used to declare objects that emulate mathematical real numbers, that can be out of the range of integer values as well as fractional values. These numbers are represented by the following notation: + or - number.number[E + or - number]. The range of the REAL type is implementation-defined. Aldec's simulator specifies it from -1.0E308 to +1.0E308. All of the basic mathematical functions such as addition, subtraction, multiplication, and division can be applied to operands of the REAL type.

Syntax

```
type REAL is range -1.0E308 to 1.0E308;
```

TIME Type

TIME is a numeric type used to specify time values represented by integer numbers of the implementation-defined range followed by the specific time unit. The range is specified in Aldec's simulator from -2, 47, 83, 47 to +2, 47, 83, 47. Time unit must be one of the following:

- fs femtosecond
- ps = 1000 fs picosecond
- ns = 1000 ps nanosecond
- us = 1000 ns microsecond
- ms = 1000 us millisecond
- sec = 1000 ms second
- min = 60 sec minute
- hr = 60 min hour

Syntax

```
type TIME is range -2147483647 to 2147483647
units
  fs;
  ps = 1000 fs;
  ns = 1000 ps;
  us = 1000 ns;
  ms = 1000 us;
  sec = 1000 ms;
  min = 60 sec;
  hr = 60 min;
end units;
```

DELAY_LENGTH Subtype

DELAY_LENGTH is a subtype of the TIME type of the range from 0 to TIME'HIGH. The DELAY_LENGTH subtype is used to represent the value of the simulation time.

Syntax

```
subtype DELAY_LENGTH is TIME range 0 fs to TIME'HIGH;
```

NATURAL Subtype

NATURAL is a numeric subtype of the type INTEGER of the range from 0 to INTEGER'HIGH. The subtype values represent mathematical natural numbers. All of the basic mathematical functions such as addition, subtraction, multiplication, and division can be applied to operands of the NATURAL subtype.

Syntax

```
subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;
```

POSITIVE Subtype

POSITIVE is a numeric subtype of the type INTEGER of the range from 1 to INTEGER'HIGH. The subtype values represent mathematical natural numbers that are greater than 0. All of the basic mathematical functions such as addition, subtraction, multiplication, and division can be applied to operands of the POSITIVE subtype.

Syntax

```
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;
```

STRING Type

STRING is a one-dimensional unconstrained array type whose values are of type CHARACTER .

Syntax

```
type STRING is array ( POSITIVE range <> ) of CHARACTER;
```

BIT_VECTOR Type

BIT_VECTOR is a one-dimensional unconstrained array type whose values are of type BIT .

Syntax

```
type BIT_VECTOR is array ( NATURAL range <> ) of BIT;
```

INTEGER_VECTOR Type (VHDL 2008)

INTEGER_VECTOR is a one-dimensional unconstrained array indexed by values of type NATURAL whose values are of type INTEGER.

Complete description: Language Reference Manual IEEE 1076-2008, § 5.3.2.3.

Syntax

```
subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;
type INTEGER_VECTOR is array ( NATURAL range `<> ) of INTEGER;
```

Examples

```
subtype samples_t is INTEGER_VECTOR( 1 to 30 );
```

BOOLEAN_VECTOR Type (VHDL 2008)

BOOLEAN_VECTOR is a one-dimensional unconstrained array indexed by values of type NATURAL whose values are of type BOOLEAN.

Complete description: Language Reference Manual IEEE 1076-2008, § 5.3.2.3.

Syntax

```
type BOOLEAN_VECTOR is array ( NATURAL range `<> ) of BOOLEAN;
```

Examples

```
variable switch_state : BOOLEAN_VECTOR( 31 downto 0 );
```

REAL_VECTOR Type (VHDL 2008)

REAL_VECTOR is a one-dimensional unconstrained array indexed by values of type NATURAL whose values are of type REAL.

Complete description: Language Reference Manual IEEE 1076-2008, § 5.3.2.3.

Syntax

```
type REAL_VECTOR is array ( NATURAL range <> ) of REAL;
```

Examples

```
subtype coeff_array is REAL_VECTOR( 0 to 63 );
```

TIME_VECTOR Type (VHDL 2008)

TIME_VECTOR is a one-dimensional unconstrained array indexed by values of type NATURAL whose values are of type TIME.

Complete description: Language Reference Manual IEEE 1076-2008, § 5.3.2.3.

Syntax

```
type TIME_VECTOR is array ( NATURAL range <> ) of TIME;
```

Examples

```
signal watchdog_timer : TIME_VECTOR( 3 downto 0 );
```

FILE_OPEN_KIND Type

FILE_OPEN_KIND is a type for opening files consisting of three values that can be specified to the OPEN_KIND parameter of the FILE_OPEN procedure. The values specify the mode of the file access.

Value	File open access mode
READ_MODE	read-only result access mode
WRITE_MODE	write-only result access mode
APPEND_MODE	write-only result access mode; information is appended to the end of existing file

Syntax

```
type FILE_OPEN_KIND is (
  READ_MODE,
  WRITE_MODE,
  APPEND_MODE );
```

FILE_OPEN_STATUS Type

FILE_OPEN_STATUS is an enumeration type whose values can be returned through the `Status` parameter of the `FILE_OPEN` procedure. It consists of 4 values that indicate the result of the `FILE_OPEN` procedure call.

Value	Result of the procedure call
OPEN_OK	the call to FILE_OPEN was successful
STATUS_ERROR	the file object already has an external file associated with it in attempt to read the external file; the file does not exist;
NAME_ERROR	in attempt to write or append to an external file: the file cannot be created
MODE_ERROR	the external file cannot be opened with the requested access type specified in the Open Kind parameter of the FILE_OPEN procedure

Syntax

```
type FILE_OPEN_STATUS is (
  OPEN_OK,
  STATUS_ERROR,
  NAME_ERROR,
  MODE_ERROR );
```

Functions

NOW Function

NOW is a function provided by the package STANDARD that returns the current simulation time, Tc. The current simulation time values are of subtype DELAY_LENGTH .

Syntax

```
impure function NOW return DELAY_LENGTH;
```

STD.TEXTIO

Overview

The Textual Input and Output (TextIO) package contains declarations of types and subprograms that support reading from and writing to formatted text files. These text files are ASCII files of any desired format that is supported by a host computer. The TextIO package treats these text files as files of lines, which are strings terminated by a carriage return. Package TEXTIO may not be modified by the user.

Types

LINE Type

LINE is an access type designating a STRING value which is a line to write to a file or a line that has just been read from the file. The LINE type is the basic unit upon which all TextIO operations are performed.

Syntax

```
type LINE is access STRING;
```

For a variable L of type LINE, L'LENGTH attribute gives the current length of the line, whether the line is being read or written. In particular, the expression L'LENGTH= 0 is true precisely when the end of the current line has been reached while reading the text file.

TEXT Type

TEXT is a file type representing files of variable-length text strings (ASCII records).

Syntax

```
type TEXT is file of STRING;
```

Function ENDFILE is defined for files of type TEXT by the implicit declaration of this function as part of the declaration of the file type:

```
function ENDFILE ( file F : FT ) return BOOLEAN;
```

SIDE Type

SIDE is a type consisting of two values: RIGHT and LEFT used for justifying output data within fields. The type is used in the JUSTIFIED parameter of the WRITE procedure that is provided in the TEXTIO package.

Syntax

```
type SIDE is ( RIGHT, EFT )
```

WIDTH Subtype

WIDTH is a subtype representing NATURAL values used for specifying widths of output fields. The subtype is used in the FIELD parameter of the WRITE procedure that is provided in the TEXTIO package.

Syntax

```
subtype WIDTH is NATURAL;
```

Standard Text Files

Package TEXTIO contains two declarations of standard text files: one for opening files in a READ_MODE, and one for opening files in a WRITE_MODE.

Syntax

```
file INPUT: TEXT open READ_MODE is "STD_INPUT";
file OUTPUT: TEXT open WRITE_MODE is "STD_OUTPUT"
```

Procedures

READLINE Procedure

The READLINE procedure reads an entire line from an ASCII text file specified in the F parameter to an object designated by the variable of type LINE specified in the L parameter. Procedure READLINE causes the next line to be read from the file and returns as the value of parameter L an access value that designates an object representing that line. The representation of a line does not contain the representation of the end of line. If the file specified in the call to the READLINE procedure is not open or is open with an access mode different that read-only, the error is reported.

Syntax

```
procedure READLINE( file F: TEXT; L: out LINE );
```

WRITELINE Procedure

The WRITELINE procedure writes an entire line designated by a variable specified in the L parameter to the text ASCII file specified in the F parameter. The procedure causes the current line designated by parameter L to be written to the file and returns with the value of parameter L designating a null string. If the file specified in the call to the WRITELINE procedure is not open or is open with an access mode other than write-only, the error is reported.

Syntax

```
procedure WRITELINE( file F: TEXT; L: inout LINE );
```

READ Procedures

READ procedures extract data from the beginning of the string value designated by the L parameter, and then modify the value so that it designates the remaining part of the line on exit. The READ procedures are declared for the following types of the VALUE parameter: BIT, BIT_VECTOR, BOOLEAN, CHARACTER, INTEGER, REAL, STRING, and TIME.

For each predefined data type there are two READ procedures. The first one has three parameters: L - the line from which to read VALUE - the value read from the line GOOD - a BOOLEAN flag indicating a success or failure of the READ procedure call; this allows a process to recover from unexpected discrepancies in input format. The other form of read operation has only the L and VALUE parameters. The read operation causes an error to occur if the requested type cannot be read into VALUE from LINE.

It is an error if the file object is not open or is open in the access mode other than READ_MODE. An error will occur when a READ operation is performed on file F if ENDFILE(F) returns TRUE at that point. ENDFILE is a function implicitly declared as part of the TEXT type declaration.

The type of the VALUE parameter specifies the way in which characters are composed into a string representation of the given type. The READ procedures defined for types other than CHARACTER or STRING begin by skipping leading whitespace characters. A whitespace character is defined as a space, a nonbreaking space, or a horizontal tabulation character (SP, NBSP, or HT). For all READ procedures characters are then removed from L and composed into a string representation of the value of the specified type. If during the process a character is encountered that cannot be part of the value (according to the lexical rules specified in IEEE Standard VHDL Language Reference Manual p. 13.2), then the character is not removed from L, and is not added to the string representation of the value. The process of removing characters and composing a string is then stopped. The READ procedures for types INTEGER and REAL also accept a leading sign, but additionally, there can be no space between a sign and the rest of the literal. The READ procedures for the types STRING and BIT_VECTOR also terminate acceptance when the VALUE'LENGTH characters have been accepted. The accepted characters are then interpreted as a string representation of the specified type.

READ does not succeed if the sequence of characters removed from L is not a valid string representation of a value of the specified type or, in case of types STRING and BIT_VECTOR, if the sequence does not contain VALUE'LENGTH characters.

The table below contains definitions of the string representation of the value for each data type.

String Representations of the Predefined Data Types

Value Type	Description
BIT	formed by a single character, either 0 or 1; no leading or trailing quotations are present
BIT_VECTOR	formed by a sequence of characters, either 0 or 1; no leading or trailing quotations are present
BOOLEAN	formed by an identifier, either FALSE or TRUE
CHARACTER	formed by a single character
INTEGER	formed as decimal literal, with the addition of an optional leading sign; no spaces can occur between the sign and the remainder of the value; leading and trailing zeroes are written as necessary to meet the requirements of the FIELD and DIGITS parameters, and they are accepted during a read
REAL	formed as decimal literal, with the addition of an optional leading sign; no spaces can occur between the sign and the reminder of the value; the decimal point is present; an exponent may optionally be present; when the exponent is present it is written as the lowercase "e" character; leading and trailing zeroes are written as necessary to meet the requirements of the FIELD and DIGITS parameters, and they are accepted during a read
STRING	formed by a sequence of characters, one for each element of the STRING; no quotation characters are present
TIME	formed by an optional decimal literal composed following the rules for INTEGER and REAL, one or more blanks, and an identifier that is a unit of type TIME

Syntax

```
procedure READ( L:inout LINE; VALUE: out BIT; GOOD : out BOOLEAN );
procedure READ( L:inout LINE; VALUE: out BIT );
procedure READ( L:inout LINE; VALUE: out BIT_VECTOR; GOOD : out BOOLEAN );
procedure READ( L:inout LINE; VALUE: out BIT_VECTOR );
procedure READ( L:inout LINE; VALUE: out BOOLEAN; GOOD : out BOOLEAN );
procedure READ( L:inout LINE; VALUE: out BOOLEAN );
procedure READ( L:inout LINE; VALUE: out CHARACTER; GOOD : out BOOLEAN );
procedure READ( L:inout LINE; VALUE: out CHARACTER );
procedure READ( L:inout LINE; VALUE: out INTEGER; GOOD : out BOOLEAN );
procedure READ( L:inout LINE; VALUE: out INTEGER );
procedure READ( L:inout LINE; VALUE: out REAL; GOOD : out BOOLEAN );
procedure READ( L:inout LINE; VALUE: out REAL );
procedure READ( L:inout LINE; VALUE: out STRING; GOOD : out BOOLEAN );
procedure READ( L:inout LINE; VALUE: out STRING );
procedure READ( L:inout LINE; VALUE: out TIME; GOOD : out BOOLEAN );
procedure READ( L:inout LINE; VALUE: out TIME );
```

WRITE Procedures

WRITE procedures append data to the end of the string value designated by parameter L. The L continues designating the entire line after the value is appended. The format of the appended data is defined in the String Representations of the Predefined Data Types .

For each predefined data type there is one WRITE procedure. The types are as follows: BIT, BIT_VECTOR, BOOLEAN, CHARACTER, INTEGER, REAL, STRING, and TIME.

Each WRITE procedure has at least two parameters: L - the line to which to write VALUE - the value to be written The additional parameters JUSTIFIED, FIELD, DIGITS, and UNIT control the formatting of output data. Each write operation appends data line to a line formatted within a field that is at least as long as required to represent the data value. FIELD - specifies the desired field width. Since the actual field width is always at least large enough to hold the string representation of the data value, the default value 0 for the FIELD parameter has the effect of causing the data value to be written out in a field of exactly the right width with no leading or trailing spaces. JUSTIFIED - specifies whether values are to be right- or left--justified within the field; the default is right-justified. DIGITS - specifies how many digits to the right of decimal point are to be output when writing a real number; the default value 0 indicates that the number should be output in standard form, consisting of a normalized mantissa plus exponent. If DIGITS is nonzero, then the real number is output as an integer part followed by the fractional part, using the specified number of digits. UNIT - specifies how values of type TIME are to be formatted. The value of this parameter must be one of the units declared as part of the TIME type declaration.

Syntax

```
procedure WRITE( L : inout LINE; VALUE : in BIT;
JUSTIFIED: in SIDE := right;
FIELD: in WIDTH := 0 );

procedure WRITE( L : inout LINE; VALUE : in BIT_VECTOR;
JUSTIFIED: in SIDE := right;
FIELD: in WIDTH := 0 );

procedure WRITE( L : inout LINE; VALUE : in BOOLEAN;
JUSTIFIED: in SIDE := right;
FIELD: in WIDTH := 0 );

procedure WRITE( L : inout LINE; VALUE : in CHARACTER;
JUSTIFIED: in SIDE := right;
FIELD: in WIDTH := 0 );

procedure WRITE( L : inout LINE; VALUE : in INTEGER;
JUSTIFIED: in SIDE := right;
FIELD: in WIDTH := 0 );

procedure WRITE( L : inout LINE; VALUE : in REAL;
JUSTIFIED: in SIDE := right;
FIELD: in WIDTH := 0;
DIGITS: in NATURAL := 0 );

procedure WRITE( L : inout LINE; VALUE : in STRING;
JUSTIFIED: in SIDE := right;
FIELD: in WIDTH := 0 );

procedure WRITE( L : inout LINE; VALUE : in TIME;
JUSTIFIED: in SIDE := right;
FIELD: in WIDTH := 0;
UNIT: in TIME := ns );
```

FLUSH Procedure (VHDL 2008)

The FLUSH procedure immediately writes a buffered output to a file object (specified by the argument of the procedure).

The FLUSH procedure requires the file object to be opened in the write mode. If the access mode of the file object is read-only or if the file is not opened, the simulator reports an error. If you need to write output to a file and simultaneously print it to the STDOUT, you can use the TEE procedure.

Syntax

```
procedure FLUSH ( file F: TEXT );
```

Important Notes

Frequent use of the FLUSH procedure (e.g. using `FLUSH` after each call to `WRITELINE`) may involve some simulation speed penalty due to increased disk I/O. Note that the simulator will automatically flush all files opened in WRITE mode whenever it hits a breakpoint or completes the `run` command. Therefore explicit calls to the `FLUSH` procedure are in practice needed only when you want to peek into simulator output files during long simulation runs.

Examples

```
-- ...
file results: TEXT open WRITE_MODE is "datapath.out";
-- ...
procedure WRITE_RESULTS (
  CLK : STD_LOGIC;
  RESET: STD_LOGIC;
  -- ...
) is

variable l_out : LINE;

begin
  write ( l_out, now, right, 15, ps );
  write ( l_out, CLK, right, 2 );
  write ( l_out, RESET, right, 2 );
  -- ...
  writeline ( results, l_out );
  flush ( results );
end;
```

The example shows the `FLUSH` procedure used immediately after the `WRITELINE` procedure. This guarantees that the data written to file will appear on disk immediately, without being buffered.

TEE Procedure (VHDL 2008)

The TEE procedure writes entire lines of a file of type text, similar to the WRITELINE procedure. The output appears both in the file (specified as the procedure argument) and on the simulator output (STDOUT). The file object passed to the TEE procedure must be opened in the write mode. If the access mode of the file object is read-only or if the file is not opened, the simulator reports an error.

Syntax

```
procedure TEE ( file F: TEXT; L: inout LINE );
```

Examples

```
file results: TEXT open WRITE_MODE is "datapath.out";

procedure WRITE_RESULTS (
  CLK : STD_LOGIC;
  RESET: STD_LOGIC ) is
  variable l_out : LINE;
begin
  write( l_out, now, right, 15, ps );
  write( l_out, CLK, right, 2 );
  write( l_out, RESET, right, 2 );
  -- ...
  tee ( results, l_out );
end procedure;
```

The example below shows how to use the TEE procedure to dump output to the file and print it to the STDOUT during simulation.

STD.ENV

STOP and FINISH Procedures

The STOP procedure pauses simulation. Simulation can be then resumed by the user. A call to the STOP procedure is equivalent to a call to vhpi_control function in a VHPI application and passing the vhpiStop argument to that function.

The FINISH procedures stops simulation. Simulation cannot be resumed. A call to the VHDL FINISH procedure is equivalent to a call to vhpi_control function with the vhpiStop argument. Both the STOP and FINISH procedures are implemented as VHPI foreign procedures.

Syntax

```
procedure STOP ( STATUS: INTEGER );
procedure STOP;

procedure FINISH ( STATUS: INTEGER );
procedure FINISH;
```

RESOLUTION_LIMIT Function

Function RESOLUTION_LIMIT returns the simulation resolution. The returned value is of type DELAY_LENGTH. The RESOLUTION_LIMIT function is implemented as a VHPI foreign function.

Syntax

```
function RESOLUTION_LIMIT return DELAY_LENGTH;
```

IEEE.STD_LOGIC_1164

Types

STD_ULOGIC Type

STD_ULOGIC is an enumerated type consisting of 9 values defining a 9-logic state system. The STD_ULOGIC type is the base type of a number of subtypes including the resolved subtype STD_LOGIC . The declaration of the STD_ULOGIC type defines the following values:

'U'	Uninitialized
'X'	Forcing Unknown
'0'	Forcing 0
'1'	Forcing 1
'Z'	High Impedance
'W'	Weak Unknown
'L'	Weak 0
'H'	Weak 1
'-'	Don't care

The meaning of specific values becomes clear in the context of the resolution function defined for the STD_LOGIC type.

Syntax

```
type STD_ULOGIC is ( 'U', X', 0', 1', Z', W', L', H', -' );
```

STD_LOGIC Type

STD_LOGIC is a basic type of the 9-logic state system. Technically, it is a resolved subtype of the STD_ULOGIC type that inherits all values of its base type. The 9-logic state system consists of the following values:

'U'	Uninitialized	This is the default initial value for objects of type STD_LOGIC. If no initial value is specified in the declaration of an object, the object acquires value 'U' after the initialization of simulation.
'X'	Forcing Unknown	'X' results if two or more opposing forcing values ('0' and '1') drive a signal of type STD_LOGIC.
'0'	Forcing 0	Strong logic '0' state.
'1'	Forcing 1	Strong logic '1' state.
'Z'	High impedance	High impedance state.
'W'	Weak Unknown	'W' results if two or more opposing weak values ('L' and 'H') drive a signal of type STD_LOGIC.
'L'	Weak 0	Weak logic '0' state.

'H'	Weak 1	Weak logic '1' state.
'-'	Don't care	Don't care state.

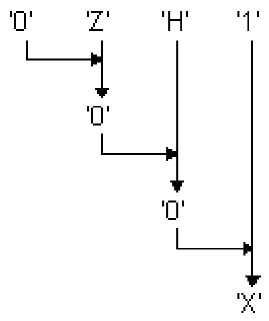
Declaration

subtype STD_LOGIC is RESOLVED STD_ULOGIC;

RESOLVED is the identifier of a resolution function. The resolution function computes values of signals of type STD_ULOGIC and its subtypes using the following table.

	'U'	**'X'**	**'0'**	**'1'**	**'Z'**	**'W'**	**'L'**	**'H'**	**'-'**
'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
'X'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'0'	'U'	'X'	'0'	'X'	'0'	'0'	'0'	'0'	'X'
'1'	'U'	'X'	'X'	'1'	'1'	'1'	'1'	'1'	'X'
'Z'	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'X'
'W'	'U'	'X'	'0'	'1'	'W'	'W'	'W'	'W'	'X'
'L'	'U'	'X'	'0'	'1'	'L'	'W'	'L'	'W'	'X'
'H'	'U'	'X'	'0'	'1'	'H'	'W'	'W'	'H'	'X'
'-'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'

The table defines the resolved value for two values of type STD_ULOGIC. If more than two values are computed by the resolution function, the resolved value is obtained according the following scheme:



STD_ULOGIC_VECTOR Type

The STD_ULOGIC_VECTOR type is a one-dimensional unconstrained array whose elements are of type STD_ULOGIC .

Syntax

```
type STD_ULOGIC_VECTOR is array ( NATURAL range <> ) of STD_ULOGIC;
```

STD_LOGIC_VECTOR Type

The STD_LOGIC_VECTOR type is a one-dimensional unconstrained array whose elements are of type STD_LOGIC .

Syntax

```
type STD_LOGIC_VECTOR is array ( NATURAL range <> ) of STD_LOGIC;
```

Additional Types

The std_logic_1164 package provides four additional subtypes of the STD_ULOGIC type. Like STD_LOGIC, all of them are resolved (using the same resolution functions RESOLVED) but their sets of values are narrowed.

Type name	Set of values
X01	'X', '0', '1'
X01Z	'X', '0', '1', 'Z'
UX01	'U', 'X', '0', '1'
UX01Z	'U', 'X', '0', '1', 'Z'

Syntax

```
subtype X01 is RESOLVED STD_ULOGIC range 'X' to '1';
subtype X01Z is RESOLVED STD_ULOGIC range 'X' to 'Z';
subtype UX01 is RESOLVED STD_ULOGIC range 'U' to '1';
subtype UX01Z is RESOLVED STD_ULOGIC range 'U' to 'Z';
```

Functions

RESOLVED Function

RESOLVED is a resolution function defined for values of type STD_ULOGIC . It is associated with all subtypes of STD_ULOGIC including STD_LOGIC .

Conversion Functions

Conversion functions enable conversion of logic values between types BIT, STD_ULOGIC and its subtypes, and one-dimensional array types composed of elements of the mentioned scalar types. All the functions require a single parameter. In case of the to-bit conversion functions (TO_BIT and TO_BITVECTOR), you can also specify the second optional parameter of type BIT, which determines how X-values ('-', 'X', 'W', 'Z', 'U') are mapped.

Function name	Parameter type	Result type
TO_BIT	STD_ULOGIC *)	BIT
TO_BITVECTOR	STD_LOGIC_VECTOR *)	BIT_VECTOR
TO_BITVECTOR	STD_ULOGIC_VECTOR *)	BIT_VECTOR
TO_STDULOGIC	BIT	STD_ULOGIC
TO_STDLOGICVECTOR	BIT_VECTOR	STD_LOGIC_VECTOR
TO_STDLOGICVECTOR	STD_ULOGIC_VECTOR	STD_LOGIC_VECTOR
TO_STDULOGICVECTOR	BIT_VECTOR	STD_ULOGIC_VECTOR
TO_STDULOGICVECTOR	STD_LOGIC_VECTOR	STD_ULOGIC_VECTOR

*) An optional second parameter is allowed which determines how X-values ('-', 'X', 'W', 'Z', 'U') are mapped. If its value is not explicitly specified in a function call, the default value '0' is assumed.

Conversions BIT-to-STD_ULOGIC is based on the following truth table:

BIT	STD_ULOGIC
'0'	'0'
'1'	'1'

Conversion STD_ULOGIC-to-BIT are based on the following truth table:

STD_ULOGIC	BIT
'U'	XMAP
'X'	XMAP
'0'	'0'
'1'	'1'
'Z'	XMAP
'W'	XMAP
'L'	'0'
'H'	'1'
'-'	XMAP

XMAP denotes the optional parameter specifying mapping of X-values.

Syntax

```
function TO_BIT ( S : STD_ULOGIC; XMAP : bit := '0' ) return BIT;
function TO_BITVECTOR ( S : STD_LOGIC_VECTOR; XMAP : BIT := '0' ) return BIT_VECTOR;
function TO_BITVECTOR ( S : STD_ULOGIC_VECTOR; XMAP : BIT := '0' ) return BIT_VECTOR;
function TO_STDULOGIC ( B : BIT ) return STD_ULOGIC;
function TO_STDLOGICVECTOR ( B : BIT_VECTOR ) return STD_LOGIC_VECTOR;
function TO_STDLOGICVECTOR ( S : STD_ULOGIC_VECTOR ) return STD_LOGIC_VECTOR;
function TO_STDULOGICVECTOR ( B : BIT_VECTOR ) return STD_ULOGIC_VECTOR;
function TO_STDULOGICVECTOR ( S : STD_LOGIC_VECTOR ) return STD_ULOGIC_VECTOR;
```

Strength Strippers

Strength strippers convert values of types BIT, STD_ULOGIC and its subtypes so that they belong to limited sets of values. The functions are also available for one-dimensional arrays composed of elements of the mentioned scalar types. All the functions require a single parameter.

Name	Parameter type	Result type	Conversion type
TO_X01	STD_LOGIC_VECTOR	STD_LOGIC_VECTOR	X01
TO_X01	STD_ULOGIC_VECTOR	STD_ULOGIC_VECTOR	X01
TO_X01	STD_ULOGIC	X01	X01
TO_X01	BIT_VECTOR	STD_LOGIC_VECTOR	01
TO_X01	BIT_VECTOR	STD_ULOGIC_VECTOR	01
TO_X01	BIT	X01	01
TO_X01Z	STD_LOGIC_VECTOR	STD_LOGIC_VECTOR	X01Z
TO_X01Z	STD_ULOGIC_VECTOR	STD_ULOGIC_VECTOR	X01Z
TO_X01	STD_ULOGIC	X01Z	X01Z
TO_X01Z	BIT_VECTOR	STD_LOGIC_VECTOR	01
TO_X01Z	BIT_VECTOR	STD_ULOGIC_VECTOR	01
TO_X01Z	BIT	X01Z	01
TO_UX01	STD_LOGIC_VECTOR	STD_LOGIC_VECTOR	UX01
TO_UX01	STD_ULOGIC_VECTOR	STD_ULOGIC_VECTOR	UX01
TO_UX01	STD_ULOGIC	UX01	UX01
TO_UX01	BIT_VECTOR	STD_LOGIC_VECTOR	01
TO_UX01	BIT_VECTOR	STD_ULOGIC_VECTOR	01
TO_UX01	BIT	UX01	01

The functions convert values using the following truth table:

	X01	**X01Z**	**UX01**	**01**
'U'	'X'	'X'	'X'	
'X'	'X'	'X'	'X'	
'0'	'0'	'0'	'0'	'0'

'1'	'1'	'1'	'1'	'1'
'Z'	'X'	'Z'	'X'	
'W'	'X'	'X'	'X'	
'L'	'0'	'0'	'0'	
'H'	'1'	'1'	'1'	
'-'	'X'	'X'	'X'	

Syntax

```
function TO_X01 ( S : STD_LOGIC_VECTOR ) return STD_LOGIC_VECTOR;
function TO_X01 ( S : STD_ULOGIC_VECTOR ) return STD_ULOGIC_VECTOR;
function TO_X01 ( S : STD_ULOGIC ) return X01;
function TO_X01 ( B : BIT_VECTOR ) return STD_LOGIC_VECTOR;
function TO_X01 ( B : BIT_VECTOR ) return STD_ULOGIC_VECTOR;
function TO_X01 ( B : BIT ) return X01;
function TO_X01Z ( S : STD_LOGIC_VECTOR ) return STD_LOGIC_VECTOR;
function TO_X01Z ( S : STD_ULOGIC_VECTOR ) return STD_ULOGIC_VECTOR;
function TO_X01Z ( S : STD_ULOGIC ) return X01Z;
function TO_X01Z ( B : BIT_VECTOR ) return STD_LOGIC_VECTOR;
function TO_X01Z ( B : BIT_VECTOR ) return STD_ULOGIC_VECTOR;
function TO_X01Z ( B : BIT ) return X01Z;
function TO_UX01 ( S : STD_LOGIC_VECTOR ) return STD_LOGIC_VECTOR;
function TO_UX01 ( S : STD_ULOGIC_VECTOR ) return STD_ULOGIC_VECTOR;
function TO_UX01 ( S : STD_ULOGIC ) return UX01;
function TO_UX01 ( B : BIT_VECTOR ) return STD_LOGIC_VECTOR;
function TO_UX01 ( B : BIT_VECTOR ) return STD_ULOGIC_VECTOR;
function TO_UX01 ( B : BIT ) return UX01;
```

Edge Detection Functions

Edge detection functions check if a falling or rising edge has occurred on a specified signal of type STD_ULOGIC or of its subtype. The returned result is of the BOOLEAN type and is equal to TRUE if an edge has been detected in the current simulation cycle.

A falling edge is detected on signal S when the following conditions are met:

- S'EVENT=TRUE
- S='0' or S='L'
- S'LAST_VALUE='1' or S'LAST_VALUE='H'

A rising edge is detected on signal S when the following conditions are met:

- S'EVENT=TRUE
- S='1' or S='H'
- S'LAST_VALUE='0' or S'LAST_VALUE='L'

Syntax

```
function RISING_EDGE ( signal S : STD_ULOGIC ) return BOOLEAN;
function FALLING_EDGE ( signal S : STD_ULOGIC ) return BOOLEAN;
```

X-Value Detection Functions

X-value detection functions return TRUE if the value of the parameter (or at least one of its elements in case of one-dimensional array parameters) is equal to an X-value, that is, 'U', 'X', 'Z', 'W', '-'.

Syntax

```
function IS_X ( S : STD_ULOGIC_VECTOR ) return BOOLEAN;
function IS_X ( S : STD_LOGIC_VECTOR ) return BOOLEAN;
function IS_X ( S : STD_ULOGIC ) return BOOLEAN;
```

Logic Operators

AND Logical Operator

Overloaded functions implementing the and operator extend its operability on types defined in the std_logic_1164 package. The following table shows allowed types of the operands and returned result.

Left operand type	Right operand type	Result type
STD_ULOGIC	STD_ULOGIC	UX01
STD_LOGIC_VECTOR	STD_LOGIC_VECTOR	STD_LOGIC_VECTOR
STD_ULOGIC_VECTOR	STD_ULOGIC_VECTOR	STD_ULOGIC_VECTOR

The result is computed on the basis of the following truth table:

	'U'	**'X'**	**'0'**	**'1'**	**'Z'**	**'W'**	**'L'**	**'H'**	**'-'**
'U'	'U'	'U'	'0'	'U'	'U'	'U'	'0'	'U'	'U'
'X'	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'
'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'
'1'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'Z'	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'
'W'	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'
'L'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'
'H'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'-'	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'

Syntax

```
function "AND" ( L : STD_ULOGIC; R : STD_ULOGIC ) return UX01;
function "AND" ( L, R : STD_LOGIC_VECTOR ) return STD_LOGIC_VECTOR;
function "AND" ( L, R : STD_ULOGIC_VECTOR ) return STD_ULOGIC_VECTOR;
```

NAND Logical Operator

Overloaded functions implementing the nand operator extend its operability on types defined in the std_logic_1164 package. The table below shows allowed types of the operands and returned result.

Left operand type	Right operand type	Result type
STD_ULOGIC	STD_ULOGIC	UX01
STD_LOGIC_VECTOR	STD_LOGIC_VECTOR	STD_LOGIC_VECTOR
STD_ULOGIC_VECTOR	STD_ULOGIC_VECTOR	STD_ULOGIC_VECTOR

The result is computed on the basis of the truth tables defined for operators and and not.

Syntax

```
function "NAND" ( L : STD_ULOGIC; R : STD_ULOGIC ) return UX01;
function "NAND" ( L, R : STD_LOGIC_VECTOR ) return STD_LOGIC_VECTOR;
function "NAND" ( L, R : STD_ULOGIC_VECTOR ) return STD_ULOGIC_VECTOR;
```

OR Logical Operator

Overloaded functions implementing the or operator extend its operability on types defined in the std_logic_1164 package. The table below shows allowed types of the operands and returned result.

Left operand type	Right operand type	Result type
STD_ULOGIC	STD_ULOGIC	UX01
STD_LOGIC_VECTOR	STD_LOGIC_VECTOR	STD_LOGIC_VECTOR
STD_ULOGIC_VECTOR	STD_ULOGIC_VECTOR	STD_ULOGIC_VECTOR

The result is computed on the basis of the following truth table:

	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'-'
'U'	'U'	'U'	'U'	'1'	'U'	'U'	'U'	'1'	'U'
'X'	'U'	'X'	'X'	'1'	'X'	'X'	'X'	'1'	'X'
'0'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'
'Z'	'U'	'X'	'X'	'1'	'X'	'X'	'X'	'1'	'X'
'W'	'U'	'X'	'X'	'1'	'X'	'X'	'X'	'1'	'X'
'L'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'H'	'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'
'-'	'U'	'X'	'X'	'1'	'X'	'X'	'X'	'1'	'X'

Syntax

```
function "OR" ( L : STD_ULOGIC; R : STD_ULOGIC ) return UX01;
function "OR" ( L, R : STD_LOGIC_VECTOR ) return STD_LOGIC_VECTOR;
function "OR" ( L, R : STD_ULOGIC_VECTOR ) return STD_ULOGIC_VECTOR;
```

NOR Logical Operator

Overloaded functions implementing the nor operator extend its operability on types defined in the std_logic_1164 package. The table below shows allowed types of the operands and returned result.

Left operand type	Right operand type	Result type
STD_ULOGIC	STD_ULOGIC	UX01

STD_LOGIC_VECTOR	STD_LOGIC_VECTOR	STD_LOGIC_VECTOR
STD_ULOGIC_VECTOR	STD_ULOGIC_VECTOR	STD_ULOGIC_VECTOR

The result is computed on the basis of the truth tables defined for operators or and not.

Syntax

```
function "NOR" ( L : STD_ULOGIC; R : STD_ULOGIC ) return UX01;
function "NOR" ( L, R : STD_LOGIC_VECTOR ) return STD_LOGIC_VECTOR;
function "NOR" ( L, R : STD_ULOGIC_VECTOR ) return STD_ULOGIC_VECTOR;
```

XOR Logical Operator

Overloaded functions implementing the xor operator extend its operability on types defined in the std_logic_1164 package. The following table shows allowed types of the operands and returned result.

Left operand type	Right operand type	Result type
STD_ULOGIC	STD_ULOGIC	UX01
STD_LOGIC_VECTOR	STD_LOGIC_VECTOR	STD_LOGIC_VECTOR
STD_ULOGIC_VECTOR	STD_ULOGIC_VECTOR	STD_ULOGIC_VECTOR

The result is computed on the basis of the following truth table:

	'U'	**'X'**	**'0'**	**'1'**	**'Z'**	**'W'**	**'L'**	**'H'**	**'-'**
U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
X'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
0'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
1'	'U'	'X'	'1'	'0'	'X'	'X'	'1'	'0'	'X'
Z'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
W'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
L'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
H'	'U'	'X'	'1'	'0'	'X'	'X'	'1'	'0'	'X'
-'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'

Syntax

```
function "XOR" ( L : STD_ULOGIC; R : STD_ULOGIC ) return UX01;
function "XOR" ( L, R : STD_LOGIC_VECTOR ) return STD_LOGIC_VECTOR;
function "XOR" ( L, R : STD_ULOGIC_VECTOR ) return STD_ULOGIC_VECTOR;
```

XNOR Logical Operator

Overloaded functions implementing the `xnor` operator extend its operability on types defined in the std_logic_1164 package. The table below shows allowed types of the operands and returned result.

Left operand type	Right operand type	Result type
STD_ULOGIC	STD_ULOGIC	UX01
STD_LOGIC_VECTOR	STD_LOGIC_VECTOR	STD_LOGIC_VECTOR
STD_ULOGIC_VECTOR	STD_ULOGIC_VECTOR	STD_ULOGIC_VECTOR

The result is computed on the basis of the truth tables defined for operators `xor` and `not`.

Syntax

```
function "XNOR" ( L : STD_ULOGIC; R : STD_ULOGIC ) return UX01;
function "XNOR" ( L, R : STD_LOGIC_VECTOR ) return STD_LOGIC_VECTOR;
function "XNOR" ( L, R : STD_ULOGIC_VECTOR ) return STD_ULOGIC_VECTOR;
```

NOT Logical Operator

Overloaded functions implementing the not operator extend its operability on types defined in the std_logic_1164 package. The following table shows allowed types of the operand and returned result.

Operand type	Result type
STD_ULOGIC	UX01
STD_LOGIC_VECTOR	STD_LOGIC_VECTOR
STD_ULOGIC_VECTOR	STD_ULOGIC_VECTOR

The result is computed on the basis of the following truth table:

'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'-'
'U'	'X'	'1'	'0'	'X'	'X'	'1'	'0'	'X'

Syntax

```
function "NOT" ( L : STD_ULOGIC ) return UX01;
function "NOT" ( L : STD_LOGIC_VECTOR ) return STD_LOGIC_VECTOR;
function "NOT" ( L : STD_ULOGIC_VECTOR ) return STD_ULOGIC_VECTOR;
```

IEEE.STD_LOGIC_ARITH

Types

SIGNED Type

The SIGNED type is a one-dimensional unconstrained array whose elements are of type STD_LOGIC . The SIGNED type is interpreted as a binary number in 2's complement notation. The leftmost array's element value is interpreted as a sign bit of a binary number. Though the array's elements are of type STD_LOGIC, they should not have any of X values: 'X', 'Z', 'W', 'U' and '-'. If X values are found in operands in arithmetic expressions then the warning will be reported and the result of the operation will be an array of 'X' values. The 'L' and 'H' values of the SIGNED type object elements are interpreted as '0' and '1' respectively when the object is used as an argument of arithmetic operations.

Syntax

```
type SIGNED is array ( NATURAL range <> ) of STD_LOGIC;
```

UNSIGNED Type

The SIGNED type is a one-dimensional unconstrained array whose elements are of type STD_LOGIC. The UNSIGNED type is interpreted as a binary number when used as an operand in arithmetic expressions. Though the array's elements are of type STD_LOGIC, they should not have any of X values: 'X', 'Z', 'W', 'U' and '-'. If X values are found in arguments of arithmetic expressions then the warning will be reported and the result of the operation will be an array of 'X' values. The 'L' and 'H' values of the UNSIGNED type object elements are interpreted as '0' and '1' respectively when the object is used as an argument of arithmetic operations.

Syntax

```
type UNSIGNED is array ( NATURAL range <> ) of STD_LOGIC;
```

SMALL_INT Subtype

The IEEE.std_logic_arith package provides an additional subtype SMALL_INT of the type INTEGER with the set of values narrowed to the '0' and '1' only.

Syntax

```
subtype SMALL_INT is INTEGER range 0 to 1;
```

Functions

Add Arithmetic Operator

The "+" arithmetic operator adds two operands of the types listed in the table below, interpreting them as binary numbers. If any of elements of the operand array has one of the X values: 'X', 'U', 'W', 'Z', '-', the operator returns an array of 'X' values and reports the warning: "There is a 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).". The 'L' and 'H' values of the operand's array elements are interpreted as '0' and '1', respectively.

Left operand type	Right operand type	Result type
SIGNED	SIGNED	SIGNED
UNSIGNED	UNSIGNED	UNSIGNED
SIGNED	UNSIGNED	SIGNED
UNSIGNED	SIGNED	SIGNED
UNSIGNED	INTEGER	UNSIGNED
INTEGER	UNSIGNED	UNSIGNED
SIGNED	INTEGER	SIGNED
INTEGER	SIGNED	SIGNED
UNSIGNED	STD_ULOGIC	UNSIGNED
STD_ULOGIC	UNSIGNED	UNSIGNED
SIGNED	STD_ULOGIC	SIGNED
STD_ULOGIC	SIGNED	SIGNED
UNSIGNED	UNSIGNED	STD_LOGIC_VECTOR
SIGNED	SIGNED	STD_LOGIC_VECTOR
SIGNED	UNSIGNED	STD_LOGIC_VECTOR
UNSIGNED	SIGNED	STD_LOGIC_VECTOR
UNSIGNED	INTEGER	STD_LOGIC_VECTOR
INTEGER	UNSIGNED	STD_LOGIC_VECTOR
SIGNED	INTEGER	STD_LOGIC_VECTOR
INTEGER	SIGNED	STD_LOGIC_VECTOR
UNSIGNED	STD_ULOGIC	STD_LOGIC_VECTOR
STD_ULOGIC	UNSIGNED	STD_LOGIC_VECTOR
SIGNED	STD_ULOGIC	STD_LOGIC_VECTOR
STD_ULOGIC	SIGNED	STD_LOGIC_VECTOR

Syntax

```
function "+" ( L: l_type; R: r_type ) return result_type;
```

Subtract Arithmetic Operation

The "-" arithmetic operator subtracts right operand from the left operand for the operands of the types listed in the table below, interpreting them as binary numbers. If any of elements of the operand array has one of the X values: 'X', 'U', 'W', 'Z', '-', the operator returns an array of 'X' values and reports the warning: "There is a

'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).". The 'L' and 'H' values of the operand's array elements are interpreted as '0' and '1', respectively.

Left operand type	Right operand type	Result type
SIGNED	SIGNED	SIGNED
UNSIGNED	UNSIGNED	UNSIGNED
SIGNED	UNSIGNED	SIGNED
UNSIGNED	SIGNED	SIGNED
UNSIGNED	INTEGER	UNSIGNED
INTEGER	UNSIGNED	UNSIGNED
SIGNED	INTEGER	SIGNED
INTEGER	SIGNED	SIGNED
UNSIGNED	STD_ULOGIC	UNSIGNED
STD_ULOGIC	UNSIGNED	UNSIGNED
SIGNED	STD_ULOGIC	SIGNED
STD_ULOGIC	SIGNED	SIGNED
UNSIGNED	UNSIGNED	STD_LOGIC_VECTOR
SIGNED	SIGNED	STD_LOGIC_VECTOR
SIGNED	UNSIGNED	STD_LOGIC_VECTOR
UNSIGNED	SIGNED	STD_LOGIC_VECTOR
UNSIGNED	INTEGER	STD_LOGIC_VECTOR
INTEGER	UNSIGNED	STD_LOGIC_VECTOR
SIGNED	INTEGER	STD_LOGIC_VECTOR
INTEGER	SIGNED	STD_LOGIC_VECTOR
UNSIGNED	STD_ULOGIC	STD_LOGIC_VECTOR
STD_ULOGIC	UNSIGNED	STD_LOGIC_VECTOR
SIGNED	STD_ULOGIC	STD_LOGIC_VECTOR
STD_ULOGIC	SIGNED	STD_LOGIC_VECTOR

Syntax

```
function "-" ( L: l_type; R: r_type ) return result_type;
```

Unary Operators

Unary operators require only one operand of the type SIGNED, or UNSIGNED, that is interpreted as a binary number. If any of elements of the operand array has one of the X values: 'X', 'U', 'W', 'Z', '-', the operator returns an array of 'X' values and reports the warning: "There is a 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es)." The 'L' and 'H' values of the operand's array elements are interpreted as '0' and '1' respectively.

There are three unary operators:

- "+" - returns operand value
- "-" - returns the result of subtracting operand from 0
- ABS - returns absolute value of an operand

Syntax

```
function "+" ( L: SIGNED ) return SIGNED;
function "+" ( L: UNSIGNED ) return UNSIGNED;
function "+" ( L: SIGNED ) return STD_LOGIC_VECTOR;
function "+" ( L: UNSIGNED ) return STD_LOGIC_VECTOR;
function "-" ( L: SIGNED ) return SIGNED;
function "-" ( L: SIGNED ) return STD_LOGIC_VECTOR;
function "ABS" ( L: SIGNED ) return SIGNED;
function "ABS" ( L: SIGNED ) return STD_LOGIC_VECTOR;
```

Multiplication Operator

The "*" arithmetic operator multiplies two operands of the types listed in the table below, interpreting them as binary numbers. If any of elements of the operand array has one of the X values: 'X', 'U', 'W', 'Z', '-', the operator returns an array of 'X' values and reports the warning: "There is a 'U'|'X'|'W' |'Z'|'-' in an arithmetic operand, the result will be 'X'(es).". The 'L' and 'H' values of the operand's array elements are interpreted as '0' and '1', respectively.

Left operand type	Right operand type	Result type
SIGNED	SIGNED	SIGNED
UNSIGNED	UNSIGNED	UNSIGNED
SIGNED	UNSIGNED	SIGNED
UNSIGNED	SIGNED	SIGNED
UNSIGNED	UNSIGNED	STD_LOGIC_VECTOR
SIGNED	SIGNED	STD_LOGIC_VECTOR
SIGNED	UNSIGNED	STD_LOGIC_VECTOR
UNSIGNED	SIGNED	STD_LOGIC_VECTOR

Syntax

```
function "*" ( L: l_type; R: r_type ) returns result_type;
```

Comparison Operators

The operators of comparison are used in conditional expressions comparing two operands of the types listed in the table below, interpreting them as binary numbers. The operands must be of the same length. The result of comparison is always of the type BOOLEAN. If any of elements of the operand array has one of the X values: 'X', 'U', 'W', 'Z', '-', the operator returns an array of 'X' values and reports the warning: "There is a 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).". The 'L' and 'H' values of the operand's array elements are interpreted as '0' and '1', respectively.

Left operand type	Right operand type	Result type
SIGNED	SIGNED	BOOLEAN
UNSIGNED	UNSIGNED	BOOLEAN
SIGNED	UNSIGNED	BOOLEAN
UNSIGNED	SIGNED	BOOLEAN
UNSIGNED	INTEGER	BOOLEAN

INTEGER	UNSIGNED	BOOLEAN
SIGNED	INTEGER	BOOLEAN
INTEGER	SIGNED	BOOLEAN

There are six comparison operators:

"<"	less than operator
"<="	less than or equal to operator
">"	greater than operator
">="	greater than or equal to operator
"="	equal to operator
"/="	not equal operator

Syntax

```
function "<" ( L: l_type; R: r_type ) return BOOLEAN;
function "<=" ( L: l_type; R: r_type ) return BOOLEAN;
function ">" ( L: l_type; R: r_type ) return BOOLEAN;
function ">=" ( L: l_type; R: r_type ) return BOOLEAN;
function "=" ( L: l_type; R: r_type ) return BOOLEAN;
function "/=" ( L: l_type; R: r_type ) return BOOLEAN;
```

Shift Operators

The shift operators shift left or right the ARG operand of the types UNSIGNED or SIGNED by the number of binary positions specified by the COUNT operand of the type UNSIGNED. The ARG operand is interpreted as a binary number. If any of elements of the operand array has one of the X values: 'X', 'U', 'W', 'Z', '-', the operator returns an array of 'X' values and reports the warning: "There is a 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es)."

There are two shift operators:

- "SHL" - shifts left the ARG operand by the number of positions specified by the COUNT operand; the COUNT number of the rightmost positions are filled with the '0' values
- "SHR" - shifts right the ARG operand by the number of positions specified by the COUNT operand; the COUNT number of the leftmost positions are filled with the '0' values for the ARG operand of the type UNSIGNED, and the value equal to the sign bit value for the ARG operand of the type SIGNED.

Syntax

```
function SHL( ARG: UNSIGNED; COUNT: UNSIGNED ) return UNSIGNED;
function SHL( ARG: SIGNED; COUNT: UNSIGNED ) return SIGNED;
function SHR( ARG: UNSIGNED; COUNT: UNSIGNED ) return UNSIGNED;
function SHR( ARG: SIGNED; COUNT: UNSIGNED ) return SIGNED;
```

Conversion Operators

The conversion operators use ARG operand of the types INTEGER, UNSIGNED, SIGNED or STD_ULOGIC, and return its value converted to the result type. The table below shows the available conversion operators together with the ARG operand types and the result types. The CONV_UNSIGNED, CONV_SIGNED and CONV_STD_LOGIC_VECTOR operators have additionally the second operand SIZE of the type INTEGER that specifies the size of the result array. The ARG operand is interpreted as a binary number. If any of elements of the ARG operand array has one of the X values: 'X', 'U', 'W', 'Z', '-' then the operator returns an array of 'X' values and reports the warning: "There is a 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).". The 'L' and 'H' values of the operand's array elements are interpreted as '0' and '1', respectively.

NOTE: The CONV_INTEGER function returns the 0 result and reports the error message: "ARG is too large in CONV_INTEGER" if the size of the ARG operand array is greater then 31 for the UNSIGNED type and 32 for the SIGNED type of the operand.

The CONV_INTEGER function returns the result of the subtype SMALL_INT for the ARG of the type STD_ULOGIC.

Conversion operator	**ARG operand type**	**Result type**
CONV_INTEGER	INTEGER	INTEGER
CONV_INTEGER	UNSIGNED	INTEGER
CONV_INTEGER	SIGNED	INTEGER
CONV_INTEGER	STD_ULOGIC	SMALL_INT
CONV_UNSIGNED	INTEGER	UNSIGNED
CONV_UNSIGNED	UNSIGNED	UNSIGNED
CONV_UNSIGNED	SIGNED	UNSIGNED
CONV_UNSIGNED	STD_ULOGIC	UNSIGNED
CONV_SIGNED	INTEGER	SIGNED
CONV_SIGNED	UNSIGNED	SIGNED
CONV_SIGNED	SIGNED	SIGNED
CONV_SIGNED	STD_ULOGIC	SIGNED
CONV_STD_LOGIC_VECTOR	INTEGER	STD_LOGIC_VECTOR
CONV_STD_LOGIC_VECTOR	UNSIGNED	STD_LOGIC_VECTOR
CONV_STD_LOGIC_VECTOR	SIGNED	STD_LOGIC_VECTOR
CONV_STD_LOGIC_VECTOR	STD_ULOGIC	STD_LOGIC_VECTOR

Syntax

```vhdl
function "CONV_INTEGER" ( ARG: operand_type ) returns result_type;
function "CONV_UNSIGNED" ( ARG: operand_type; SIZE: INTEGER ) returns result_type;
function "CONV_SIGNED" ( ARG: operand_type; SIZE: INTEGER ) returns result_type;
function "CONV_STD_LOGIC_VECTOR" ( ARG: operand_type ) returns result_type;
```

Extension Functions

The package provides two extension functions that allow extending the ARG operand's length to the value provided by the SIZE operand. Additional array's elements have the '0' values in case of the EXT function, and the sign bit values in case of the SXT function. The ARG operand is of the type STD_LOGIC_VECTOR and the returned result is of the type STD_LOGIC_VECTOR(SIZE-1 downto 0). If the SIZE parameter value is less than the ARG parameter length, the leftmost elements of the ARG parameter will be truncated. If any of elements of the ARG operand array has one of the X values: 'X', 'U', 'W', 'Z', '-' then the operator returns an array of 'X' values and reports the warning: "There is a 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).". The 'L' and 'H' values of the operand's array elements are interpreted as '0' and '1', respectively. Negative values of SIZE are interpreted as zeros.

- "EXT" - zero extend STD_LOGIC_VECTOR(ARG) TO SIZE
- "SXT" - sign extend STD_LOGIC_VECTOR(ARG) TO SIZE

Syntax

```
function EXT( ARG: STD_LOGIC_VECTOR; SIZE: INTEGER ) return STD_LOGIC_VECTOR;
function SXT( ARG: STD_LOGIC_VECTOR; SIZE: INTEGER ) return STD_LOGIC_VECTOR;
```