The VHDL Golden Reference Guide

DOULOS

Version 1.1, December 1995

DOULOS
Church Hatch,
22 Market Place,
Ringwood.
Hampshire.
BH24 1AW
England.

Tel (+44) (0)1425 471223
Fax (+44) (0)1425 471573

Email info@doulos.co.uk

Preface

The VHDL Golden Reference Guide is a compact quick reference guide to the VHDL language, its syntax, semantics, synthesis and application to hardware design.

The VHDL Golden Reference Guide is not intended as a replacement for the IEEE Standard VHDL Language Reference Manual. Unlike that document, the Golden Reference guide does not offer a complete, formal description of VHDL. Rather, it offers answers to the questions most often asked during the practical application of VHDL, in a convenient reference format.

Nor is The VHDL Golden Reference Guide intended to be an introductory tutorial. Information is presented here in a terse reference format, not in the progressive and sympathetic manner necessary to learn a subject as complex as VHDL. However, acknowledging that those already familiar with computer languages may wish to use this guide as a VHDL text book, a brief informal introduction to the subject is given at the start.

The main feature of The VHDL Golden Reference Guide is that it embodies much practical wisdom gathered over many VHDL projects. It does not only provide a handy syntax reference; there are many similar books which perform that task adequately. It also warns you of the most common language errors, gives clues where to look when your code will not compile, alerts you to synthesis issues, and gives advice on improving your coding style.

The VHDL Golden Reference Guide was developed to add value to the Doulos range of VHDL training courses, and also to complement VHDL PaceMaker, the VHDL Computer Based Training package from Doulos.

Using This Guide

The main body of this guide is organised alphabetically. Each section is indexed by a key term which appears prominently at the top of each page. Often you can find the information you want by flicking through the guide looking for the appropriate key term. If that fails, there is a full index at the back.

Most of the information in this guide is organised around the VHDL syntax headings, but there are additional special sections on Coding Standards, Design Flow, Errors, Reserved Words and VHDL 93, and also listings of the standard packages Standard, TEXTIO, Std_logic_1164 and Numeric_std.

If you are new to VHDL, you should start by reading A Brief Introduction to VHDL, which follows overleaf.

The Index

Bold index entries have corresponding pages in the main alphabetical reference section. The remaining index entries are followed by a list of appropriate page references in the main alphabetical reference section, given in order of importance.

Key To Notation Used To Define VHDL Syntax

The syntax definitions are written to look like examples whereever possible, but it has been necessary to introduce some extra notation. In brief, square brackets [] enclose optional items, three dots ... means repetition, and curly brackets {} enclose comments. *ItalicNames* represent parts of the syntax defined elsewhere. A full description of the notation follows:

Curly brackets {} enclose comments that are not part of the VHDL syntax being defined, but give you further information about the syntax definition.

Syntax enclosed in square brackets [] is optional (except in the definition of a signature, where square brackets are part of the VHDL syntax!)

... means zero or more repetitions of the preceding item or line, or means a list, as follows:

Item ... means zero or more repetitions of the Item.
, ... means repeat in a comma separated list (e.g. A, B, C).
; ... means repeat in a semicolon separated list.
| ... means repeat in a bar separated list.

There must be at least one item in the list. There is no **, ;** or **|** at the end of the list, unless it is given explicitly (as in **; ... ;**).

Underlined syntax belongs to the VHDL'93 language, but not to VHDL'87. (For the sake of clarity, underlining has been omitted where words contain the underscore character.)

words in lower case letters are reserved words, built into the VHDL language (e.g. entity)

Capitalised Words (not in italics) are VHDL identifiers, i.e. user defined or pre-defined names that are not reserved identifiers (e.g. TypeName, BlockLabel).

Italic Words are syntactic categories, i.e. the name of a syntax definition given in full elsewhere. A syntactic category can be either defined on the same page, defined on a separate page, or one of the two special categories defined below.

Italics = indicates a syntactic category which is defined and used on the same page.

Special syntactic categories:

SomethingExpression = *Expression*, where the *Something* gives information about the meaning of the expression (e.g. *TimeExpression*).

Condition = *Expression*, where the type of the expression is Boolean.

A Brief Introduction To VHDL

The following paragraphs give a brief technical introduction to VHDL suitable for the reader with no prior knowledge of the language. As will be evident from these paragraphs, VHDL uses a lot a specialised technical jargon!

Background

The letters VHDL stand for the VHSIC (Very High Speed Integrated Circuit) Hardware Description Language. VHDL is a language for describing the behaviour and structure of electronic circuits, and is an IEEE standard (1076).

VHDL is used to simulate the functionality of digital electronic circuits at levels of abstraction ranging from pure behaviour down to gate level, and is also used to synthesize (i.e. automatically generate) gate level descriptions from more abstract (Register Transfer Level) descriptions. VHDL is commonly used to support the high level design (or language based design) process, in which an electronic design is verified by means of thorough simulation at a high level of abstraction before proceeding to detailed design using automatic synthesis tools.

VHDL became an IEEE standard in 1987, and this version of the language has been widely used in the electronics industry and academia. The standard was revised in 1993 to include a number of significant improvements.

The Language

In this section as in the rest of the guide, words given in *Capitalised Italics* are technical terms whose definitions may be found in the main body of this guide.

An hierarchical portion of a hardware design is described in VHDL by an *Entity* together with an *Architecture*. The *Entity* defines the interface to the block of hardware (i.e. the inputs and outputs), whilst the *Architecture* defines its internal structure or behaviour. An *Entity* may possess several alternative *Architectures.*

Hierarchy is defined by means of *Components*, which are analogous to chip sockets. A *Component* is *Instantiated* within an *Architecture* to represent a copy of a lower level hierarchical block. The association between the *Instance* of the *Component* and the lower level *Entity* and *Architecture* is only made when the complete design hierarchy is assembled before simulation or synthesis (analogous to plugging a chip into a chip socket on a printed circuit board). The selection of which *Entity* and *Architecture* to use for each *Component* is made in the *Configuration*, which is like a parts list for the design hierarchy.

The structure of an electronic circuit is described by making *Instances* of *Components* within an *Architecture*, and connecting the *Instances* together using *Signals*. A *Signal* represents an electrical connection, a wire or a bus. A *Port Map* is used to connect *Signals* to the *Ports* of a *Component Instantiation*, where a *Port* represents a pin.

Each *Signal* has a *Type*, as does every value in VHDL. The *Type* defines both a set of values and the set of operations that can be performed on those values. A *Type* is often defined in a *Package*, which is a piece of VHDL containing definitions which are common to several *Entities*, *Architectures*, *Configurations* or other *Packages*. Individual wires are often represented as *Signals* of type Std_logic, which are defined in the package *Std_logic_1164*, another IEEE standard.

The behaviour of an electronic circuit is described using *Processes* (which represent the leaves in the hierarchy tree of the design). Each *Process* executes concurrently with respect to all other *Processes,* but the statements inside a process execute in sequential order and are in many ways similar to the statements in a software programming language. A *Process* can be decomposed into named *Procedures* and *Functions*, which can be given parameters. Common *Procedures* and *Functions* can be defined in a *Package*.

Compilation

VHDL source code is usually typed into a text file on a computer. That text file is then submitted to a VHDL compiler which builds the data files necessary for simulation or synthesis. The proper jargon for the steps performed by the compiler are Analysis, which checks the VHDL source for errors and puts the VHDL into a *Library*, and Elaboration, which links together the *Entities* and *Architectures* of the hierarchy.

```
library IEEE;
use IEEE.Std_logic_1164.all;
entity EntName is
  port (P1, P2: in Std_logic;
        P3: out Std_logic_vector(7 downto 0));
end EntName;
architecture ArchName of EntName is
  component CompName
    port (P1: in  Std_logic;
          P2: out Std_logic);
  end component;
  signal SignalName, SignalName2: Std_logic := 'U';
begin
  P: process (P1,P2,P3) -- Either sensitivity list or wait statements!
    variable VariableName, VarName2: Std_logic := 'U';
  begin
    SignalName <= Expression after Delay;
    VariableName := Expression;
    ProcedureCall(Param1, Param2, Param3);
    wait for Delay;
    wait until Condition;
    wait;
    if Condition then
      -- sequential statements
    elsif Condition then
      -- sequential statements
    else
      -- sequential statements
    end if;
    case Selection is
    when Choice1 =>
      -- sequential statements
    when Choice2 | Choice3 =>
      -- sequential statements
    when others =>
      -- sequential statements
    end case;
    for I in A'Range loop
      -- sequential statements
    end loop;
  end process P;
  SignalName <= Expr1 when Condition else Expr2;
  InstanceLabel: CompName port map (S1, S2);
  L2: CompName port map (P1 => S1, P2 => S2);
  G1: for I in A'Range generate
    -- concurrent statements
  end generate G1;
end ArchName;
```

```
package PackName is
  type Enum is (E0, E1, E2, E3);
  subtype Int is Integer range 0 to 15;
  type Mem is array (Integer range <>) of
            Std_logic_vector(7 downto 0);
  subtype Vec is Std_logic_vector(7 downto 0);

  constant C1: Int := 8;
  constant C2: Mem(0 to 63) := (others => "11111111");

  procedure ProcName (ConstParam: Std_logic;
                      VarParam: out Std_logic;
               signal SigParam: inout Std_logic);

  function "+" (L, R: Std_logic_vector)
                      return Std_logic_vector;
end PackName;

package body PackName is

  procedure ProcName (ConstParam: Std_logic;
                      VarParam: out Std_logic;
               signal SigParam: inout Std_logic) is
    -- declarations
  begin
    -- sequential statements
  end ProcName;

  function "+" (L, R: Std_logic_vector)
                      return Std_logic_vector is
    -- declarations
  begin
    -- sequential statements
    return Expression;
  end "+";
end PackName;

configuration ConfigName of EntityName is
  for ArchitectureName
    for Instances: ComponentName
      use LibraryName.EntityName(ArchName);
    end for;
  end for;
end ConfigName;
```

This quick reference syntax summary does not follow the notational conventions used in the rest of the Guide.

The VHDL Golden Reference Guide

Alphabetical Reference Section

Access

A data type which allows dynamic memory allocation, equivalent to pointers in C or Pascal. Used to model large memories. The type Line in package TEXTIO is an access type. An incomplete type declaration is used to permit recursively defined data structures, e.g. linked lists.

Syntax

```
type NewName is access DataType;

type IncompleteTypeName;
```

Where

See Declaration

Rules

- Only variables can be of access type, and they must point to a value allocated dynamically using **new** (not to another variable).
- An access variable is initialised to the value **null**.
- A procedure DEALLOCATE(Ptr) is implicitly defined and can be called to release the storage allocated using **new**.

Gotchas!

VHDL suffers from dangling pointers. If you copy a pointer then deallocate the memory, the copied pointer still points to the now deallocated location.

Synthesis

Not synthesizable.

Tips

Can be used to reduce the amount of memory needed to simulate a large memory by only allocating host computer memory when needed.

Example

```
type Link; -- Incomplete type declaration
type Item is record
  Data: Std_logic_vector(7 downto 0);
  NextItem: Link;
end record;
type Link is access Item;
variable StartOfList, Ptr: Link;

-- Add item to start of list
Ptr := new Item;  -- Allocate storage
Ptr.Data := "01010101";
Ptr.NextItem := StartOfList; -- Link item into list
StartOfList := Ptr;
```

```
-- Delete entire list
while StartOfList /= null loop
  Ptr := StartOfList.NextItem;
  DEALLOCATE(StartOfList);
  StartOfList := Ptr;
end loop;
```

See Also

New, Type, Null, Record

Aggregate

A way to write a value for any array or record.

Syntax

```
([Choices =>] Expression, ...)

Choices = Choice | ...
Choice = {either}
ConstantExpression
Range
others     {the last choice}
```

Where

See Expression, Target

Rules

- An aggregate must give a value for every element of the array or record.
- The two forms of syntax (ordered list or explicitly named choices) can be mixed, but the ordered values must come before the named choices.

Gotchas!

Aggregates frequently need to be qualified to disambiguate their type (see example below).

Synthesis

Many synthesis tools do not allow aggregates as targets of assignments.

Tips

The aggregate (**others** => *Expression*) is a very useful way of setting all the elements of an array to the same value; you do not even have to know how big the array is! For example, to set the value of a parameter of an unconstrained array type.

Example

```
('0', '1', '0', '1')
(1, 2, 3, 4, 5)
(1 => A, 2 => B, 3 => C)
(1, 2, 3, others => 4)
(others => 'Z')
(A, B, C) := D;            -- Aggregate as the target of an assignment
T'(others => '0')          -- Qualified expression
(others => T'(others => '0'))
```

See Also

Array, Record, Expression, Range

Lets you give an alternative name for almost anything. Particularly useful for renaming a slice of an array, as it avoids the need to define a new signal or variable. Also allows one package to inherit procedures and functions from another package by aliasing them.

Syntax

```
alias AliasName [: Datatype] is Name [Signature];

Signature = [TypeName, ...] return TypeName
```

Where

See Declaration

Rules

- The AliasName may be an identifier, character or operator.
- An aliased procedure, function or enumeration literal must be identified unambiguously by a signature, which identifies the parameter types and return type (to allow for overloading).

Gotchas!

The *DataType* and the *Name* being aliased must both be static, so an aliased slice name must have a static index constraint.

Synthesis

Not supported by many synthesis tools.

Example

```
function F (A, B: Std_logic_vector) return BOOLEAN is
  alias P1: Std_logic_vector(1 to A'LENGTH) is A;
  alias P2: Std_logic_vector(1 to B'LENGTH) is B;
  -- Alias is used to create 2 local vectors with the same range
begin
  for I in P1'RANGE loop
    if P1(I) = P2(I) then
      ...

alias ">" is
  F[Std_logic_vector, Std_logic_vector] return BOOLEAN;
```

See Also

Constant

Architecture

Defines the internal view of a block of hardware, i.e. the functionality, behaviour or structure of the hardware. Belongs with an entity, which defines the interface. An entity may have several alternative architectures.

Syntax

```
architecture ArchitectureName of EntityName is
  Declarations...
begin
  ConcurrentStatements...
end [architecture] [ArchitectureName];
```

Where

See (VHDL) File

Rules

All the architectures of a particular entity must have different names, but the architectures of two different entities can have the same name.

Gotchas!

It is easy to forget the **begin**, or put it in the wrong place!

Example

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

architecture BENCH of TEST_MUX4 is
  subtype V2 is STD_LOGIC_VECTOR(1 downto 0);
  -- Component declaration...
  component MUX4
    port (SEL, A, B, C, D: in  V2;
          F              : out V2);
  end component;
  -- Internal signal...
  signal SEL, A, B, C, D, F: V2;
begin
  P: process
  begin
    SEL <= "00";
    wait for 10 NS;
    SEL <= "01";
    wait for 10 NS,
    SEL <= "10";
    wait for 10 NS,
    SEL <= "11";
    wait for 10 NS;
    wait;
  end process P;
```

```
    --  Concurrent assignments...
    A <= "00";
    B <= "01";
    C <= "10";
    D <= "11";
    --  Component instantiation...
    M: MUX4 port map (SEL, A, B, C, D, F);
  end BENCH;
```

See Also

Entity, Configuration, Concurrent Statement

Array

A data type which consists of a vector or a multi-dimensional set of values of the same base type. Can be used to describe RAMs, ROMs, FIFOs, or any regular multi-dimensional structure.

Syntax

```
type NewName is                        {unconstrained}
  array (IndexTypeName range <>, ...) of DataType;

type NewName is                        {constrained}
  array (Range, ...) of DataType;
```

Where

See Declaration

Rules

- The base type *DataType* must not be an unconstrained array type.
- A signal or variable cannot be an unconstrained array, unless it is a generic, port or parameter.

Synthesis

Some synthesis tools do not support multi-dimensional arrays, only support arrays of bits or arrays of vectors, or do not permit ports to be arrays of arrays.

Tips

- Large arrays should be variables or constants, rather than signals. A large signal array would be inefficient for simulation.
- The values within an array can be read or written using an indexed name or a slice name.

Example

```
subtype Word is Std_logic_vector(15 downto 0);
type Mem is array (0 to 2**12-1) of Word;
variable Memory: Mem := (others => Word'(others=>'U'));
...
if MemoryRead then
  Data <= Memory(To_Integer(Address));
elsif MemoryWrite then
  Memory(To_Integer(Address)) := Data;
end if;
```

See Also

Range, Name, String, Type

A sequential or concurrent statement used to write out a message when an exception occurs. If the condition is False, the simulator writes out a report to the screen or log file. The simulator may be instructed to halt if the severity is above a particular level.

Syntax

```
[Label:] assert Condition
  [report StringExpression]
  [severity Expression];
```

Where

```
entity-begin-<HERE>-end
architecture-begin-<HERE>-end
block-begin-<HERE>-end
generate-begin-<HERE>-end
```

See Sequential Statement

Rules

- Sequential statements can be labelled in VHDL'93, but not in VHDL'87.
- The severity *Expression* must be of type Severity_level, which has the values Note, Warning, Error, Failure. The default severity is Error.

Gotchas!

Check carefully the sense of the *Condition*. The message is written when the *Condition* is False!

Synthesis

Assertions do not represent hardware. Synthesis tools ignore them or give a warning.

Example

```
assert not (Reset = '0' and Set = '0')
  report "R-S conflict" severity Failure;
assert Outputs = ExpectedOutputs
  report "Outputs differ from expected response";
```

See Also

Report, TEXTIO

Attribute

A user defined attribute is used to attach arbitrary information to a specific part of a VHDL description for use by downstream tools (e.g. synthesis or device fitting). Any attribute not recognized by a particular tool is ignored. Each attribute must be both declared and specified as shown below.

Syntax

```
{declaration}
attribute AttributeName: TypeName;

{specification}
attribute AttributeName of Name [Signature]: Class is
                                           Expression;

Signature = [TypeName, ...] return TypeName
Class = {either} signal type function architecture {etc}
```

Where

See Declaration

An attribute specification must be in the region in which the *Name* is declared. Attributes of an entity, architecture, configuration or package must be specified inside that region. Neither attribute declaration nor specification is allowed in a *Package Body*.

Rules

- *Name Signature* may be replaced by **others**, **all**, or a list.
- A procedure, function or enumeration literal must be identified unambiguously by a *Signature*, which identifies the parameter types and return type (to allow for overloading).

Example

```
attribute Pin_number: Positive;
attribute Pin_number of Clk: signal is 1;
attribute Enum_encoding: String;
attribute Enum_encoding of State: type is "11 00 01 10";
```

See Also

Attribute Name, Group

Attribute Name

An attribute gives extra informific part of a VHDL description, and can be usation about a specer defined or predefined. User defined attributes are constants, whereas predefined attributes can be constants, functions or signals.

Syntax

```
Name[Signature]'AttributeName[(Expression)]

Signature = [TypeName, ...] return TypeName
```

Where

See Expression

Rules

A procedure, function or enumeration literal must be identified unambiguously by a signature, which identifies the parameter types and return type (to allow for overloading).

Predefined Type Attributes

T is an enumeration, integer, floating or physical type or subtype

T'BASE	The base type of T. Only allowed as prefix to another attribute
T'LEFT	Left bound of T
T'RIGHT	Right bound of T
T'LOW	Lower bound of T
T'HIGH	Upper bound of T
T'ASCENDING	TRUE if range of T is **to**, FALSE if **downto**
T'IMAGE(X)	The string representation of X in T
T'VALUE(X)	The value of type T whose string representation is X
T'POS(X)	Position number of X in T
T'VAL(X)	Value with position number X in T
T'SUCC(X)	Successor = T'VAL(T'POS(X)+1)
T'PRED(X)	Predecessor = T'VAL(T'POS(X)-1)
T'LEFTOF(X)	Value to the left of X in T
T'RIGHTOF(X)	Value to the right of X in T

Predefined Array Attributes

A is an array signal, variable, constant, type or subtype

A'LEFT[(N)]	Left bound of Nth index range
A'RIGHT[(N)]	Right bound of Nth index range

A'LOW[(N)]	Lower bound of Nth index range
A'HIGH[(N)]	Upper bound of Nth index range
A'RANGE[(N)]	Range of Nth index from left to right
A'REVERSE_RANGE[(N)]	Range of Nth index from right to left
A'LENGTH[(N)]	The number of values in the Nth index range
A'ASCENDING[(N)]	TRUE if Nth index range of A is **to**, FALSE if **downto**

Predefined Signal Attributes

S is a signal

S'DELAYED[(T)]	A signal with the value that signal S had at time NOW-T
S'STABLE[(T)]	A signal which is TRUE if and only if no event has occurred on signal S for time T
S'QUIET[(T)]	A signal which is TRUE if and only if no transaction has occurred on S for time T
S'TRANSACTION	A signal of type BIT which toggles whenever there is a transaction on S (A signal assignment creates a transaction. A transaction that causes a change in value is an event)
S'EVENT	TRUE if and only if there is an event on S in the current delta
S'ACTIVE	TRUE if and only if there is a transaction on S in the current delta
S'LAST_EVENT	The time since the last event on S
S'LAST_ACTIVE	The time since the last transaction on S
S'LAST_VALUE	The value of S before the last event
S'DRIVING	FALSE if and only if the value of the driver of S in the current process is **null**
S'DRIVING_VALUE	The value of the driver for S in the current process

Predefined General Attributes

E is the name of just about anything!

E'SIMPLE_NAME	The string representation of the name of E
E'INSTANCE_NAME	The string representation of the hierarchical path name of E, including the names of instantiated entities. Of the form

	":ent(arch):componentlabel@ent(arch):componentlabel@ent(arch):thing" or ":lib:pack:thing"
• E'PATH_NAME	The string representation of the hierarchical path name of E, excluding the names of instantiated entities. Of the form ":ent:componentlabel:componentlabel:thing" or ":lib:pack:thing"

Gotchas!

- Type and array attributes have subtly different meanings. T'HIGH gives the maximum value of integer or enumeration type T, but A'HIGH gives the maximum value of the index of array A, not the maximum value that can be stored in A.
- The attribute S'EVENT is not a signal, so should not be used in a context where a signal is needed to trigger a process. The expression not S'STABLE can be used instead. For example, the following will not work:

```
wait until A'EVENT or B'EVENT; -- waits forever!
```

Synthesis

Attributes of enumeration types should not be used for synthesis, nor should the attributes POS, VAL, SUCC, PRED, LEFTOF, RIGHTOF. Many synthesis tools do not support these particular attributes, and attributes of enumeration types can become invalid during optimization.

Tips

Type and array attributes should be used wherever possible to make code easier to read and maintain. The RANGE attribute is particularly useful for looping through arrays.

Example

```
type T is (A, B, C, D, E);
subtype S is T range D downto B;
S'LEFT = D
S'RIGHT = B
S'LOW = B
S'HIGH = D
S'BASE'LEFT = A
T'ASCENDING = TRUE
S'ASCENDING = FALSE
T'IMAGE(A) = "a"
T'VALUE("E") = E
T'POS(A) = 0
S'POS(B) = 1
T'VAL(4) = E
S'SUCC(B) = C
S'PRED(C) = B
S'LEFTOF(B) = C
S'RIGHTOF(C) = B
signal A: STD_LOGIC_VECTOR(7 downto 0);
A'LEFT = 7
A'RIGHT = 0
A'LOW = 0
A'HIGH = 7
A'RANGE = 7 downto 0
A'REVERSE_RANGE = 0 to 7
A'LENGTH = 8
A'ASCENDING = FALSE
```

See Also

Attribute, Name, Range, Signal

A concurrent statement used to group together concurrent statements, and make local declarations. Guarded blocks provide an alternative (but little used) way to write Register Transfer Level descriptions. The execution of guarded signal assignments within the block is controlled by the guard expression at the top.

Syntax

```
BlockLabel: block [(GuardExpression)] [is]
  [Generic;
  [GenericMap;]]
  [Port;
  [PortMap;]]
  Declarations...
begin
  ConcurrentStatements...
end block [BlockLabel];
```

Where

```
architecture-begin-<HERE>-end
block-begin-<HERE>-end
generate-begin-<HERE>-end
```

Gotchas!

Some commercial synthesis tools do not support blocks.

Tips

It is not necessary to learn and use blocks and related syntax such as guarded signal assignments. It is generally more efficient for simulation to use processes instead.

Example

```
signal P, Q, R: STD_LOGIC;
...
Logic: block
  port (A, B: in  STD_LOGIC;
        F   : out STD_LOGIC);
  port map (A => P, B => Q, F => R);
begin
  F <= A nand B;
end block Logic;

Sync: block (Rising_edge(Clock))
begin
  Q <= guarded D; -- This assignment occurs on the clock edge
  QB <= not Q;    -- This assignment occurs when Q changes
end block Sync;
```

See Also

Signal Assignment, Concurrent Statement, Disconnect

A sequential statement which conditionally executes one branch only, depending on the value of the expression at the top.

Syntax

```
[Label:] case Expression is
when Choices =>
  SequentialStatements...
when Choices =>
  SequentialStatements...
... {any number of when parts}
end case [Label];

Choices = Choice | Choice | ...
Choice = {either}
ConstantExpression
Range
others      {the last branch}
```

Where

See Sequential Statement

Rules

- The *Expression* must not be enclosed in parenthesis.
- The type of the *Expression* must be enumeration, integer, physical, or a one dimensional array.
- Every case of the *Expression* must be covered once and only once by the *Choices*.

Gotchas!

The **|** is not the "or" operator; the *Choices* are not or'd together!

Synthesis

- Assignments within case statements generally synthesize to multiplexers.
- Incomplete assignments (i.e. where outputs remain unassigned for certain input conditions) in unclocked processes synthesize to transparent latches. Incomplete assignments in clocked processes synthesize to recirculation around registers.

Example

```
case ADDRESS is
when 0 =>                 --  Select a single value
  A <= '1';
when 1 =>
  A <= '1';               --  More than one statement in a branch
  B <= '1';
when 2 to 15 =>           --  Select a range of ADDRESS values
  C <= '1';
when 16 | 20 | 24 =>      --  Pick out several ADDRESS values
  B <= '1';
  C <= '1';
  D <= '1';
when others =>            --  Mop up the rest
  null;
end case;
```

See Also

If, Select, Null, Range

Coding Standards

Coding standards are divided into two categories. Lexical coding standards, which control text layout, naming conventions and commenting, are intended to improve readability and ease of maintenance. Synthesis coding standards, which control VHDL style, are intended to avoid common synthesis pitfalls and find synthesis errors early in the design flow.

The following lists of coding standards will need to be modified according to the choice of tools and personal preferences.

Lexical Coding Standards

- Limit the contents of each VHDL source file to one entity and its architectures, one package and its package body, or several related configurations.
- Source file names should relate to the file contents (e.g. EntityName.vhdl, PackageName.vhdl or configs.vhdl).
- Adopt a naming convention for architectures (e.g. Reference, Behaviour, RTL, GateLevel, TestBench).
- Write only one declaration or statement per line.
- Use indentation as shown in the examples.
- Be consistent about the case of keywords (e.g. lower case), predefined names (e.g. upper case), and user defined names (e.g. first letter a capital).
- User defined names should be meaningful and informative, although local names (e.g. loop parameters) may be terse.
- Label all processes and concurrent assignments, both as documentation and to aid debugging.
- Write comments to explain (not duplicate) the VHDL code. It is particularly important to comment interfaces (e.g. generics, ports, parameters and packages).
- Use constants and attributes wherever possible, instead of directly embedding literal numbers and strings in declarations and statements.
- Write use clauses at the top of each entity, package or configuration, to make dependencies easy to find.

Synthesis Coding Standards

- Partition the design into small functional blocks, and use a behavioural style for each block. Avoid gate level descriptions except for critical parts of the design.
- Have a well defined clocking strategy, and implement that strategy explicitly in VHDL (e.g. single clock, multi-phase clocks, gated clocks, multiple clock domains). Ensure that clock and reset signals in VHDL are clean (i.e. not generated from combinational logic or unintentionally gated).
- Have a well defined (manufacturing) testing strategy, and code up the VHDL appropriately (e.g. all flipflops resettable, test access from external pins, no functional redundancy).

- Every VHDL process should conform to one of the standard synthesizable process templates (see *Process*).
- VHDL processes describing combinational and latched logic must have all of their inputs in the sensitivity list.
- Combinational processes must not contain incomplete assignments, i.e. all outputs must be assigned for all combinations of input values.
- Processes describing combinational and latched logic must not contain feedback, i.e. signals and variables assigned as outputs from the process must not be read as inputs to the process.
- Clocked processes with a sensitivity list must have only the clock and any asynchronous control inputs (usually reset or set) in the sensitivity list.
- Clocked processes without a sensitivity list must have only one wait statement, of the form **wait until clock_edge_expression**, as the first executable statement in the process.
- Avoid unwanted latches. Unwanted latches are caused by incomplete assignments in an unclocked process.
- Avoid unwanted flipflops. Flipflops are synthesized when signals are assigned in a clocked process, or when variables assigned in a clocked process retain their value between process executions (and thus between clock cycles).
- All internal state registers must be resettable, in order that the Register Transfer Level and gate level descriptions can be reset into the same known state for verification. (This does not apply to pipeline or synchronization registers.)
- For finite state machines and other sequential circuits with unreachable states (e.g. a 4 bit decade counter has 6 unreachable states), if the behaviour of the hardware in such states is to be controlled, then the behaviour in all 2^N possible states must be described explicitly in VHDL, including the behaviour in unreachable states. This allows safe state machines to be synthesized.
- Avoid delays in signal assignments, except where necessary to solve the problem of delta delay clock skew at register transfer level.
- Do not initialise signals or variables to values other than 'U' or 'X'. Take great care when using types with no uninitialised value (e.g. integers), because the VHDL simulation will not reveal initialisation problems.
- Do not write VHDL code which relies on the order of values within an enumeration type, because the order might be changed during optimization. Such code is also harder to maintain.
- Signals and variables of type INTEGER should have a range constraint, otherwise they will synthesize to 32 bit busses.
- Check carefully any VHDL code which uses dynamic indexing (i.e. an index expression containing signals or variables), loop statements, or arithmetic operators, because such code can synthesize to large numbers of gates which can be hard to optimize.

A component is analogous to a chip socket; it gives an indirect way to use one hierarchical block within another. A component is instantiated within an architecture, and is associated with a (lower level) entity and architecture during elaboration using information from a configuration.

Syntax

```
component ComponentName [is]
  [Generic;]
  [Port;]
end component [ComponentName];
```

Where

```
package-<HERE>-end
architecture-is-<HERE>-begin-end
block-<HERE>-begin-end
generate-<HERE>-begin-end
```

See Declaration

Rules

For default configuration, the component name must match the name of the corresponding entity to be used in its place, and generics and ports must also match in name, mode and type.

Synthesis

A component without a corresponding design entity is synthesized as a black box.

Tips

In VHDL'93, components are not necessary. It is possible instead to directly instantiate an entity within an architecture.

Example

```
component Counter
  generic (N: INTEGER);
  port (Clock, Reset, Enable: in Std_logic;
        Q: buffer Std_logic_vector (N-1 downto 0));
end component;
```

See Also

Instantiation, Generic, Port, Generic Map, Port Map, Configuration, Entity

Concurrent Statement

A statement which is concurrent with respect to all other such statements. The following are concurrent statements:

- Process
- Instantiation
- Signal assignment
- Generate
- Assert
- Procedure call
- Block

Where

```
entity-begin-<HERE>-end
architecture-begin-<HERE>-end
block-begin-<HERE>-end
generate-begin-<HERE>-end
```

Rules

The concurrent signal assignment, concurrent assert and concurrent procedure call are defined in terms of equivalent process statements. Thus, their labels are optional, and they may be **postponed**.

See Also

Sequential Statement

A concurrent statement which assigns one of several expressions to a signal, depending on the values of boolean conditions which are tested in sequence. Equivalent to a process containing an **if** statement.

Syntax

```
[Label:] Target <= [Options]
   Expression [after TimeExpression] when Condition else
   Expression [after TimeExpression] when Condition else
   ...
   Expression [after TimeExpression] [when Condition];

Target = {either} SignalName Aggregate

Options = {either}
guarded
transport
reject TimeExpression inertial
```

Where

```
architecture-begin-<HERE>-end
block-begin-<HERE>-end
generate-begin-<HERE>-end
```

Rules

- The reserved word **guarded** may only appear in a signal assignment within a guarded block. A guarded assignment only executes when the guard expression on the surrounding block is true.
- An *Expression* on the right hand side may be replaced by the reserved word **unaffected**.

Synthesis

Conditional signal assignments are synthesized to combinational logic. The *Expressions* on the right hand side are multiplexed onto the *Target* signal. The resulting logic will be priority encoded, because the *Conditions* are tested in sequence.

Tips

- In VHDL'93, adding a final **when** part or replacing an expression by **unaffected** both cause the target signal to retain its value under certain conditions, allowing flipflops and latches to be described as shown below.
- Conditional and selected signal assignments are a concise way to describe combinational logic in Register Transfer Level descriptions, although processes can be easier to read and maintain in some cases.
- A conditional assignment is a neat way to convert from a Boolean condition to the type Std_logic. See the first example below.

Example

```
L: Equal <= '1' when A = B else '0';

NextState <= Idle  when State = Clear else
             Start when State = Idle  else
             Stop  when State = Start else
             Clear;

Flipflop: Q <= D when Rising_edge(Clock);
Latch: Q <= D when Enable = '1' else unaffected;
```

See Also

Signal Assignment, Select, If, Block

A configuration declaration defines how the design hierarchy is linked together during elaboration, by listing the entities and architectures used in place of each component instantiation within an architecture. A configuration may also patch up differences between the names and types of generics and ports of the component and the entity.

Syntax

```
configuration ConfigurationName of EntityName is
  Declarations...          {use, attribute or group}
  for ArchitectureName     {of entity named above}
    Use...
    ConfigurationItems
  end for;
end [configuration] [ConfigurationName];

ConfigurationItems = {one or more of the following}

for BlockName              {a nested block or generate statement}
    Use...
    ConfigurationItems
end for;

for InstanceLabel, ... : ComponentName
  [use WhatToUse
    [GenericMap]
    [PortMap] ;]
  [for ArchitectureName {going down the hierarchy}
     Use...
     ConfigurationItems
  end for;]
end for;

BlockName = {either}
BlockLabel
GenerateLabel(ConstantExpression)
GenerateLabel(Range)

WhatToUse = {either}
entity EntityName[(ArchitectureName)]
configuration ConfigurationName
open                       {unconfigured}
```

Where

See (VHDL) File

Rules

- The instance labels in front of the component name can be replaced by **others** or **all**.
- Each component instance can be explicitly configured once only.
- In the absence of a configuration, component instances get configured by default to use an entity with the same name, port names and port types as the component, and to use the most recently compiled architecture.

Synthesis

Although configurations are relevant and useful for selected which entities and architectures make up the design hierarchy, many synthesis tools do not support them. Instead, write a script to synthesize the correct architectures.

Tips

- Put the configurations for a design in a separate file.
- Write a configuration for the top level test bench (initially an empty configuration). Write a configuration for an architecture only when there exists more than one entity or architecture which can be used for the components in that architecture (i.e. when there are choices to be made), and reference that configuration from a higher level configuration.

Example

```
use Work.Types.all;
entity Top is              -- Top level H/W description
  port (A, B: in Int8; F, G: out Int8);
end Top;

architecture Structure of Top is
  component Blk
    port (A: in Int8; F: out Int8);
  end component;
begin
  B1: Blk port map (A, F);
  B2: Blk port map (B, G);
end Structure;

use Work.Types.all;
entity Blk is              -- Pre-synthesis
  port (A: in Int8; F: out Int8);
end Blk;

architecture RTL of Blk is
begin
  ...
end RTL;
```

```
library IEEE;
use IEEE.Std_logic_1164.all;
entity GateLevelBlk is  -- Post-synthesis
  port (IP: in  Std_logic_vector(7 downto 0);
        OP: out Std_logic_vector(7 downto 0));
end GateLevelBlk;

architecture Synth of GateLevelBlk is
begin
  ...
end Synth;

use Work.Types.all;
configuration TopMixed of Top is
  for Structure
    for B1: Blk
      use entity Work.Blk(RTL);
    end for;
    for B2: Blk
      use entity Work.GateLevelBlk(Synth)
        port map (IP => To_Vector(A),
                  To_Int8(OP) => F);
    end for;
  end for;
end TopMixed;

use Work.Types.all;
entity Test is           -- Test bench for Top
end Test;

architecture Bench of Test is
  component Top
    port (A, B: in Int8; F, G: out Int8);
  end component;
  signal A, B, F, G: Int8;
begin
  ...
  Inst: Top port map (A, B, F, G);
end Bench;

configuration TestMixed of Test is
  for Bench
    for all: Top
      use configuration Work.TopMixed;
    end for;
  end for;
end TestMixed;
```

See Also

Configuration Specification, Component, Entity, Architecture

A configuration specification defines which entity and architecture is used in place of the instances of a single component during elaboration. A configuration specification may also patch up differences in the names and types of generics and ports for that single component.

Syntax

```
for InstanceLabel, ... : ComponentName
  use WhatToUse
    [GenericMap]
    [PortMap];

WhatToUse = {either}
entity EntityName[(ArchitectureName)]
configuration ConfigurationName
open
```

Where

```
architecture-is-<HERE>-begin-end
block-<HERE>-begin-end
generate-<HERE>-begin-end
```

See Declaration

Rules

- The instance labels in front of the component name can be replaced by **others** or **all**.
- Each component instance can be explicitly configured once only.

Tips

Configuration specifications are inflexible, because changing the configuration requires editing the architecture containing the configuration. It is usually better to use separate configuration declarations.

Example

```
-- (See Configuration)
architecture FullyBound of Top is
  component Blk
    port (A: in Int8; F: out Int8);
  end component;

  for B1: Blk use entity Work.Blk(RTL);

  for B2: Blk use entity Work.GateLevelBlk(Synth)
        port map (IP => To_Vector(A),
                  To_Int8(OP) => F);
begin
  B1: Blk port map (A, F);
  B2: Blk port map (B, G);
end FullyBound;
```

See Also

Configuration, Component, Entity, Architecture

A constant is used to give a name to a value, in order to make code easier to read and maintain.

Syntax

```
constant NewName: DataType := Expression;
```

Where

See Declaration

Rules

The value of a constant cannot be changed using an assignment statement.

Example

```
constant Width: POSITIVE := 8;
constant Mask: STD_LOGIC_VECTOR := "0001000011111";
                                        -- Range determined by length of string
```

See Also

Alias, Variable

Data Type

A data type (known as a subtype indication in the VHDL standard) appears in a declaration to identify the type used at that point. A data type can also include a constraint, which further restricts the values of the type during simulation or synthesis. A data type can also include the name of a resolution function to define the behaviour of a bus when there are conflicts between the drivers. The resolution function is called by the simulator whenever a signal is assigned.

Syntax

```
[ResolutionFunctionName] TypeName [Constraint]

Constraint = {either}
range Range                  {range constraint}
(Range, ...)                 {index constraint}
```

Rules

- The constraint must be consistent with the type; range constraint for an integer, floating, enumeration or physical type, index constraint for an unconstrained array type.
- If the value goes outside the constraint during simulation, this is an error and simulation halts with the message "Constraint Violation".

Gotchas!

- Signals and variables cannot be unconstrained arrays. You must remember the index constraint.
- A resolution function should be written such that is independent of the order of its inputs; otherwise, the results of simulation will be indeterminate!

Synthesis

- Resolution functions are ignored by most synthesis tools.
- Both range and index constraints are used to determine the widths of busses.

Example

```
BOOLEAN                          -- no constraint
INTEGER range 0 to 255
STD_LOGIC_VECTOR(7 downto 0)
RESOLVED STD_ULOGIC              -- resolution function
```

See Also

Type, Signal, Disconnect, Subtype, Range, Function

Declaration

The following parts of the language can be written in those sections of the syntax where declarations are allowed:

- Procedure
- Procedure Body
- Function
- Function Body
- Type
- Subtype
- Constant
- Signal
- Variable
- Shared Variable
- File
- Component
- Configuration Specification
- Alias
- Attribute
- Disconnect
- Use
- Group Template
- Group

Where

```
package-<HERE>-end
package body-<HERE>-end
entity-is-<HERE>-begin-end
architecture-is-<HERE>-begin-end
block-<HERE>-begin-end
generate-<HERE>-begin-end
process-<HERE>-begin-end
function-is-<HERE>-begin-end
procedure-is-<HERE>-begin-end
```

Exclusions

Region	Cannot contain Declaration
Entity	Component, Configuration, Variable
Architecture	Variable
Block	Variable
Generate	Variable

Package	Configuration, Variable, Function Body, Procedure Body
Package Body	Component, Configuration, Signal, Variable, Attribute, Disconnect
Process	Component, Configuration, Signal, Shared Variable, Disconnect
Function	Component, Configuration, Signal, Shared Variable, Disconnect
Procedure	Component, Configuration, Signal, Shared Variable, Disconnect

Declaration	**Not Allowed In Region**
Attribute	Package Body
Component	Entity, Package Body, Process, Function, Procedure
Configuration	Entity, Package, Package Body, Process, Function, Procedure
Disconnect	Package Body, Process, Function, Procedure
Signal	Package Body, Process, Function, Procedure
Function Body	Package
Procedure Body	Package
Shared Variable	Process, Function, Procedure
Variable	Entity, Architecture, Block, Generate, Package, Package Body

Design Flow

The basic flow for using VHDL and synthesis to design an ASIC or complex FPGA is shown below. Iteration around the design flow is necessary, but is not shown here. Also, the design flow must be modified according to the kind of device being designed and the specific application.

1 System analysis and specification

2 System partitioning

- 2.1 Top level block capture
- 2.2 Block size estimation
- 2.3 Initial floorplanning

3 Block level design. For each block:

- 3.1 Write Register Transfer Level VHDL
- 3.2 Synthesis coding checks
- 3.3 Write VHDL test bench
- 3.4 VHDL simulation
- 3.5 Write synthesis scripts - constraints, boundary conditions, hierarchy
- 3.6 Initial synthesis - analysis of gate count and timing

4 Chip integration. For complete chip:

- 4.1 Write VHDL test bench
- 4.2 VHDL simulation
- 4.3 Synthesis
- 4.4 Gate level simulation

5 Test generation

- 5.1 Modify gate level netlist for test
- 5.2 Generate test vectors
- 5.3 Simulate testable netlist

6 Place and route (or fit) chip

7 Post layout simulation and timing analysis

Disconnect

Defines the delay with which the drivers of a guarded signal are disconnected from the resolution function, when the signal is assigned by a guarded signal assignment. A driver is disconnected by assigning the signal the value **null**.

Syntax

```
disconnect SignalName, ... : TypeName
                                   after TimeExpression;
```

Where

```
package-<HERE>-end
entity-is-<HERE>-begin-end
architecture-is-<HERE>-begin-end
block-<HERE>-begin-end
generate-<HERE>-begin-end
```

See Declaration

Rules

- The list of signal names may be replaced by **others** or **all**.
- Each signal must be declared as a guarded signal, and must be assigned in a guarded signal assignment.

Synthesis

Not synthesizable.

Tips

Don't use guarded signals or disconnection!

Example

```
disconnect Foo: Std_logic after 10 NS;
```

See Also

Signal, Signal Assignment, Block, Data Type, Null

Defines the interface to an hierarchical block. An entity is used in combination with an architecture, which together describe the behaviour or structure of an hierarchical block of hardware (a design entity).

Syntax

```
entity EntityName is
  [Generic;]
  [Port;]
  Declarations...
[begin
  ConcurrentStatements...]      {passive processes}
end [entity] [EntityName];
```

Where

See (VHDL) File

Rules

Concurrent statements within the entity must be equivalent to passive processes, i.e. contain no signal assignments.

Synthesis

Each entity is synthesized as a separate hierarchical block, allowing you to control the hierarchy of the synthesized netlist, although some synthesis tools flatten the hierarchy by default.

Tips

- If you need two versions of an entity with different ports, you must make two different entities. Entities cannot be overloaded.
- It is not necessary to write declarations and statements inside an entity. It is usually clearer and simpler to put them in the architecture.

Example

```
library IEEE;
use IEEE.Std_logic_1164.all;
use IEEE.Numeric_std.all;

entity Counter is
  generic (N: INTEGER);
  port (Clock, Reset, Enable: in Std_logic;
        Q: buffer Std_logic_vector (N-1 downto 0));
end Counter;
```

See Also

Architecture, Generic, Port, Component, Instantiation

Enumeration

A data type defined by listing the values it can take. Each value is either a name or a character. A character is one printable character enclosed in single quotes.

Syntax

```
type NewName is (EnumLiteral, ...);

EnumLiteral = {either}
Identifier
'{One printable character}'
```

Where

See Declaration

Rules

The same value cannot appear twice in the same type, but may appear in two different enumeration types. This is called overloading.

Gotchas!

- Characters are case sensitive, e.g. 'x' is not the same as 'X'
- Don't use attributes of enumeration types with synthesis, in case the encoding is changed during optimization.

Synthesis

A user defined enumeration type of N values synthesizes to a bus of $\log_2 N$ bits. The 1st value becomes binary 0, the 2nd binary 1, the third binary 10 and so on. Finite state machine optimization changes the encoding.

Example

```
type STD_ULOGIC is
    ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
type Opcode is (Idle, Start, Stop, Clear);
```

See Also

Type, Standard, Attribute Name

This is a list of the most common VHDL errors. The Top 10 account for about 40% of all errors. The Top 20 account for about 60% of all errors.

The Top 20: Errors 1-10

- Missing or misplaced begin in architecture / process / subprogram
- Swapping <= := and =
- Missing or extra end if / end case / end loop / end process etc
- Wrong quotes around characters / strings / integers
- Incompatible types in assignments / operators
- Missing or extra ; at end of declaration / statement
- Misspelt identifier
- Undefined signal / variable / constant
- Missing library / use
- Wrong separator , / ; / : / .

The Top 20: Errors 10-20

- Missing sensitivity list / **wait** statement in process
- Sensitivity list and **wait** statement in same process
- Same name used twice
- **elseif** or **endif** (instead of **elsif** and **end if** respectively)
- Using the wrong user defined identifier
- Extra / missing / mistyped character
- Missing signals in sensitivity list
- Mismatched vector lengths
- Using a reserved identifier
- Reading **out** ports

Exit

A sequential statement which jumps straight out of a loop.

Syntax

```
[Label:] exit [LoopLabel] [when Condition];
```

Where

```
loop-<HERE>-end loop
```

See Sequential Statement

Rules

The **exit** must be inside a loop with the given LoopLabel, or inside any loop if there is no LoopLabel.

Example

```
L1: loop
  L2: for I in A'RANGE loop
    if A(I) = 'U' then
      exit L1;                    -- Leave outer loop L1
    end if;
    exit when I = N;              -- Leave inner loop L2
  end loop L2;
  ...
end loop L1;
```

See Also

Next, For Loop, While Loop, Loop

An expression calculates a value from a set of operators, names, literal values and sub-expressions. A static expression is an expression whose value can be calculated during compilation or elaboration.

Syntax

{either}
```
Expression Operator Expression
Operator Expression
(Expression)
Name
Number
PhysicalLiteral
Character
String
Aggregate
QualifiedExpression
New
FunctionCall
```

Rules

Operators are evaluated in the following order:

1st: ** abs not
2nd: * / mod rem
3rd: + - &
4th: sll srl sla sra rol ror
5th: = /= < <= > >=
6th: and or xor nand nor xnor

Example

```
A + B
not A
(A nand B) nor C
A(7 downto 0)               -- name
'0'                         -- character
(others => '0')             -- aggregate
To_integer(V)               -- function call
T'(A, B)                    -- qualified expression
new T
```

See Also

Operator, Name, Aggregate, New, Qualified Expression

File

A file is a stream of values of a specified type which can be read or written during simulation. Files belong to a special kind of data type called a file type. The only files which are commonly used are text files, read or written using package TEXTIO.

Syntax

```
type NewName is file of TypeName;

file NewName: DataType is [Mode] FileName;     {'87 only}
Mode = {either} in out                         {'87 only}

file NewName: DataType open FileKind is FileName;
FileKind = {either} Read_mode Write_mode Append_mode

FileName = StringExpression
```

Where

See Declaration

Rules

The following are defined implicitly for all file types:

```
procedure FILE_OPEN (   file F: FT;
                        External_name: in STRING;
                        Open_kind: in FILE_OPEN_KIND :=
                                      READ_MODE);
procedure FILE_OPEN (   Status: out FILE_OPEN_STATUS;
                        file F: FT;
                        External_name: in STRING;
                        Open_kind: in FILE_OPEN_KIND :=
                                      READ_MODE);
procedure FILE_CLOSE (  file F: FT);
procedure READ (        file F: FT; VALUE: out TM);
procedure WRITE (       file F: FT; VALUE: in TM);
function ENDFILE (      file F: FT) return BOOLEAN;
```

Gotchas!

- File declarations are incompatible between VHDL'87 and VHDL'93.
- Files (except those written using TEXTIO) cannot necessarily be read into another VHDL simulator, or even be read using a standard text editor!

Synthesis

Files are not synthesizable.

Tips

Always use package TEXTIO to read or write files.

Example

```
type T2 is file of T1;
file F: T2 is out "filename";              -- VHDL'87
file F: T2 open Write_mode is "filename"; -- VHDL'93
```

See Also

TEXTIO, Type

(Vhdl) File

VHDL source code is contained in text files. A file can contain one or many *Design Units*, i.e. entities, architectures, configurations, packages, or package bodies.

Syntax

{A file contains one or many *DesignUnits*}

```
DesignUnit = {both}
ContextClause ...
LibraryUnit

ContextClause = {either}
Library
Use

LibraryUnit = {either}
Entity
Architecture
Configuration
Package
PackageBody
```

Gotchas!

Library and use definitions are not global throughout a file; they must be repeated for every entity or package.

Tips

It is best to restrict the contents of a file to one entity and the associated architectures, one package and the associated package body, or a set of related configurations. This makes the VHDL source files easy to manage and avoids unnecessary re-compilation:

- Put only one entity together with its architectures in the same file.
- Do not put a package in the same file as an entity.
- Do not put a configuration in the same file as an entity.

An alternative approach is to put each design unit in a separate file. This minimizes re-compilation. but there are more files to manage.

See Also

Entity, Architecture, Configuration, Package, Library, Use

A data type representing an abstraction of a mathematical floating point number. The maximum values for the range are at least +/- 10^{38}, precision is at least 6 decimal digits.

Syntax

```
type NewName is range Range;
```

Where

See Declaration

Rules

The lower and upper bounds of the *Range* must be static floating point *Expressions*.

Gotchas!

The occurrence of an event on a floating point signal can be non-deterministic. An event is a change in value, and whether a floating point value changes depends on how it is represented in the simulator.

Synthesis

Not synthesizable.

Tips

Used for high level simulations, but cannot be used to accurately describe floating point hardware, because it cannot model accuracy, truncation, normalization etc.

Example

```
type T is range 0.0 to 1.0;
```

See Also

Number, Range, Type, Subtype, Integer

For Loop

A sequential statement used to execute a set of sequential statements repeatedly, with the loop parameter taking each of the values in the given *Range* from left to right.

Syntax

```
[LoopLabel:] for ParameterName in Range loop
  SequentialStatements...
end loop [LoopLabel];
```

Where

See Sequential Statement

Rules

- The loop parameter is implicitly declared by the loop statement itself, and only exists inside the loop.
- The loop parameter is a constant (i.e. cannot be assigned).

Synthesis

Synthesis make multiple copies of the logic implied by the statements inside the loop. Only synthesizable if the *Range* is static .

Example

```
type Opcode is (Idle, Start, Stop, Clear);
...
for I in 0 to 7 loop
  V := V xor A(I);
  for J in Opcode loop
    S <= J;
    wait for 10 NS;
  end loop;
end loop;
```

See Also

While Loop, Loop, Exit, Next, Range

Used to group together executable, sequential statements to define new mathematical or logical functions. Also used to define bus resolution functions, operators, and conversion functions between data types. When defined in a package, the function must be split into a declaration and a body.

Syntax

```
{declaration}
[Kind] function FunctionName
                         [(ParameterDeclaration; ...)]
                         return TypeName;


{body}
[Kind] function FunctionName
                         [(ParameterDeclaration; ...)]
                         return TypeName is
  Declarations...
begin
  SequentialStatements...end [function] [FunctionName];

Kind = {either} pure impure

ParameterDeclaration = {either}
constant ConstantName,... :[in] DataType [:= Expression]
signal   SignalName, ...  :[in] DataType [:= Expression]
file FileName, ... : DataType
```

Where

See Declaration

A Function Body is not allowed in a Package Declaration.

Rules

- The FunctionName may be an identifier or an operator.
- Functions cannot assign signals or variables defined outside themselves, nor can then contain **wait** statements.
- A function must execute a **return** statement.
- **Pure** functions cannot have side effects - they must do nothing but return a value.

Gotchas!

- The return type must be a name; it cannot include a constraint.
- Variables defined inside a function are initialized each time the function is called.
- The declaration and body must conform, i.e. the parameters and return type must be identical between the two.

- The function declaration ends with a ";", whereas the function body has **is** at the corresponding point in the syntax.

Synthesis

Each call to a function is synthesized as a separate block of combinational logic.

Tips

Parameters may be unconstrained arrays; you can use array attributes (e.g. 'RANGE) to find their bounds.

Example

```
package P is

function To_Std_logic_vector (Value, Width: INTEGER)
         return Std_logic_vector;

function "+" (A, B: Std_logic_vector)
              return Std_logic_vector is
end;

package body P is

function To_Std_logic_vector (Value, Width: INTEGER)
         return Std_logic_vector is
  variable V: INTEGER := Value;
  variable Result: Std_logic_vector (1 to Width);
begin
  for I in Result'REVERSE_RANGE loop
    if V mod 2 = 1 then
      Result(I) := '1';
    else
      Result(I) := '0';
    end if;
    if V >= 0 then
      V := V / 2;
    else
      V := (V - 1) / 2;
    end if;
  end loop;
  return Result;
end To_Std_logic_vector;
```

```
function "+" (A, B: Std_logic_vector)
              return Std_logic_vector is
  variable LV: Std_logic_vector(A'Length-1 downto 0);
  variable RV: Std_logic_vector(B'Length-1 downto 0);
  variable Result:
               Std_logic_vector(A'Length-1 downto 0);
  variable Carry: Std_logic := '0';
begin
  LV := A;
  RV := B;
  assert A'Length = B'Length
    report "function +: operands have different widths"
    severity Failure;
  for I in Result'Reverse_range loop
    Result(I) := LV(I) xor RV(I) xor Carry;
    Carry := (LV(I) and RV(I)) or (LV(I) and Carry) or
             (RV(I) and Carry);
  end loop;
  return Result;
end "+";

end P;
```

See Also

Function Call, Return, Procedure, Package, Type Conversion, Operator

Function Call

Calls a function, which returns a value for use in an expression.

Syntax

```
FunctionName [ ( [Formal =>] Actual, ... ) ]

Formal = {either} Name FunctionCall
Actual = Expression
```

Where

See Expression

Rules

The two forms of syntax (ordered list or explicitly named parameters) can be mixed, but the ordered list must come before the named parameters.

Tips

Use the parameter names rather than parameter order to improve readability and reduce the risk of making errors.

Example

```
B := To_Std_logic_vector(J, 2+2);
C := IEEE.Numeric_std."+" (L => A, R => B);
     -- (Equivalent to C := A + B;)
```

See Also

Function, Expression, Operator, Procedure Call

Generate

A concurrent statement used to create regular structures or conditional structures during elaboration.

Syntax

```
Label: for ParameterName in Range generate
  [Declarations...
begin]
  ConcurrentStatements...
end generate [Label];

Label: if Condition generate
  [Declarations...
begin]
  ConcurrentStatements...
end generate [Label];
```

Where

```
architecture-begin-<HERE>-end
block-begin-<HERE>-end
generate-begin-<HERE>-end
```

Rules

The *Range* and *Condition* must both be static, i.e. they cannot include signals.

Gotchas!

The Label at the beginning of the generate statement cannot be omitted.

Synthesis

Synthesis is straightforward, but not all synthesis tools support generate!

Tips

- **For...generate** is useful to replace repeated instances of the same component.
- In VHDL'87 a generate statement cannot contain declarations. To include declarations in a **generate** you would have to nest a **block** statement.

Example

```
G1: for I in 1 to Depth generate
  L: BLK port map (A(I), B(I+1));       -- Repeated instance
end generate G1;
```

```
G2: if Option = TRUE generate
  process                                          -- Optional process
  begin
    ...
  end process;
end generate;
```

See Also

Concurrent Statement, For Loop, If, Range

Used to parameterize a design entity. Different *Instances* of the same design entity can have different values for the generic parameters. Generics are given values in the generic map of an *Instance*.

Syntax

```
generic (GenericName, ... : DataType [:= Expression];
         ...);
```

Where

```
entity-is-<HERE>-port-end
component-<HERE>-port-end
block-<HERE>-generic map-port-begin-end
```

Rules

- The *Expression* gives the default value, and must be static. Only a generic with a default value can be omitted from the corresponding generic map.
- Generics are constants; they cannot be assigned new values.

Gotchas!

The generics of an entity must be duplicated in the corresponding component, to allow instances of the component to be configured implicitly via the default rules.

Synthesis

Many synthesis tools support only integer generics.

Example

```
generic (N, M: Positive;
         Mask: Std_logic_vector := "11111111");
```

See Also

Generic map, Entity, Component, Port

Generic Map

Used to define the values of generics. Usually given in an *Instance*, but may also appear in a configuration.

Syntax

```
generic map ([Formal =>] Actual, ...)

Formal = {either} Name FunctionCall
Actual = Expression
```

Where

```
Label: ComponentName <HERE> port map(-);
for-use- <HERE> port map(-)
block-generic(-); <HERE> ; port-begin-end
```

Rules

The two forms of syntax (ordered list or explicitly named choices) can be mixed, but the ordered list must come before the named choices.

Gotchas!

A generic map does not end with a semicolon!

Example

```
architecture Structure of Ent is
  component NAND2
    generic (TPLH, TPHL: TIME := 0 NS);
    port (A, B: in  STD_LOGIC;
          F   : out STD_LOGIC);
  end component;
begin
  G1: NAND2 generic map (1.9 NS, 2.8 NS)
            port map (N1, N2, N2);
  G2: NAND2 generic map (TPLH => 2 NS, TPHL => 3 NS)
            port map (N4, N5, N6);
end Structure;
```

See Also

Generic, Instantiation, Block, Configuration

Group

A group is a named collection of items. A group template defines the kind of items that can appear in a group (analogous to the type of a data object). Groups have no meaning for simulation, but (like attributes) can be used to pass information to downstream tools (e.g. synthesis).

Syntax

```
group TemplateName is (Class, ...);
Class = {either} label signal function group {etc} [<>]

group GroupName: TemplateName (Name, ...);
```

Where

See Declaration

Rules

Class <> allows a list of items at that point in the group, and must be the last item in the group template.

Example

```
architecture RTL of Ent is
  group Operations is (function, label <>);
  group Adders: Operations ("+", A1, A2, A3);
begin
  A1: X <= A + B;
  A2: Y <= C + D;
  A3: Z <= E + F;
end RTL;
```

See Also

Attribute

If

A sequential statement which executes one branch from a set of branches dependent upon the Conditions, which are tested in sequence.

Syntax

```
[Label:] if Condition then
  SequentialStatements...
[elsif Condition then
  SequentialStatements...]
... {any number of elsif parts}
[else
  SequentialStatements...]
end if [Label];
```

Where

See Sequential Statement

Gotchas!

Be careful about the spelling of **elsif** and **end if**

Synthesis

- Assignments within **if** statements generally synthesize to multiplexers.
- Incomplete assignments, where outputs remain unchanged for certain input conditions, synthesize to transparent latches in unclocked processes, and to recirculation in clocked processes.
- In some circumstances, nested **if** statements synthesize to multiple logic levels. This can be avoided by using a **case** statement instead.

Tips

A set of **elsif** branches can be used to give priority to the conditions tested first. To decode a value without giving priority to certain conditions, use a **case** statement instead.

Example

```
if C1 = '1' and C2 = '1' then
  V := not V;
  W := '0';
  if C3 = '0' then
    X := A;
  elsif C4 = '0' then
    X := B;
  else
    X := C;
  end if;
end if;
```

See Also

Case, Conditional Assignment, Generate

Instantiation

A concurrent statement used to define the design hierarchy by making a copy of a lower level design entity within an architecture. In VHDL'93, a direct instantiation of an entity bypasses the component and configuration.

Syntax

```
InstanceLabel: [component] ComponentName
                        [GenericMap] [PortMap];
InstanceLabel: entity EntityName[(ArchitectureName)]
                        [GenericMap] [PortMap];
InstanceLabel: configuration ConfigurationName
                        [GenericMap] [PortMap];
```

Where

```
architecture-begin-<HERE>-end
block-begin-<HERE>-end
generate-begin-<HERE>-end
```

Rules

An entity, architecture or configuration must be compiled into a library before the corresponding instance can be compiled. However, an instance of a component can be compiled before the corresponding design entity has even been written.

Tips

- EntityName (or ConfigurationName) usually takes the form of a selected name, because the identifier is not directly visible by default (see example).

Example

```
G1: NAND2 generic map (1.2 NS) port map (N1, N2, N3);
G2: entity WORK.Counter(RTL) port map (Clk, Rst, Count);
```

See Also

Generic map, Port map, Component, Entity, Architecture, Configuration

An integer type represents a mathematical integer. The maximum values for the range are at least +/- (2^{31} - 1).

Syntax

```
type NewName is range Range;
```

Where

See Declaration

Rules

The lower and upper bounds of the *Range* must be static integer expressions.

Gotchas!

- Integer types are not mutually compatible; they cannot be mixed unless explicit *Type Conversions* are used.
- An unconstrained INTEGER synthesizes to 32 bits, so always give a constraint for synthesis!

Synthesis

An integer range 0 to N-1 synthesizes to a bus of width $\log_2 N$ bits. The value is represented as a binary number. Negative numbers are represented in two's complement format.

Tips

It is more convenient to create a subtype of the predefined type INTEGER (e.g. subtype T is INTEGER range 0 to 7;) rather than defining a new integer type.

Example

```
type INT is range -8 to 7;
```

See Also

Number, Type, Subtype, Range, Floating, Standard

Library

Defines a library name. The library name is mapped to the pathname of a directory in the host file system by the VHDL tool, not within the language.

Syntax

```
library LibraryName, ... ;
```

Where

```
<HERE>-entity
<HERE>-architecture
<HERE>-package
<HERE>-package body
<HERE>-configuration
```

See (VHDL) File

Tips

The library name WORK is implicitly defined to mean the working library, so it is confusing to create libraries named WORK.

Example

```
library IEEE, Project;
```

See Also

(VHDL) File, Use

A sequential statement used to execute a set of sequential statements repeatedly.

Syntax

```
[LoopLabel:] loop
  SequentialStatements...
end loop [LoopLabel];
```

Where

See Sequential Statement

Gotchas!

A loop is an infinite loop (and thus an error) unless it contains an *exit* or *wait* statement.

Synthesis

Not generally synthesizable. Some tools do allow loops containing **wait** statements to describe implicit finite state machines, but this is not recommended practice.

Example

```
loop
  wait until Clock = '1';
  exit when Reset = '1';
  Div2 <= not Div2;
end loop;
```

See Also

For Loop, While Loop, Exit, Next

Name

Any VHDL "thing" is identified by its name. A selected name is commonly used to pick an item out of a library or package. An indexed name is used to pick an individual item out of an array. A slice name is used to pick out part of an array.

Syntax

```
Identifier
\ExtendedIdentifier\
"Operator"
Name.Name. ...                  {selected name}
Name(Expression, ...)           {indexed name}
Name(Range)                     {slice name}
AttributeName
```

Rules

- An identifier consists of letters, digits and underscores. The first character must be a letter. The last character must not be an underscore, nor can a name contain two adjacent underscores.
- An extended identifier consists of any printable characters.
- One name cannot have more than one meaning at any particular point in the VHDL text, with the exception of procedures, functions and enumeration literals, which may be overloaded. Inner declarations of names hide outer declarations.

Gotchas!

The direction (i.e. **to** or **downto**) of the range in a slice name must be consistent with the subtype of the thing being sliced; otherwise it is a null range, i.e. an array of length zero.

Tips

Generally, choose names which are meaningful to the reader. However, this is more important for global names than for local names. For example, G0123 is a bad name for a global reset signal, but I is an acceptable name for a loop parameter.

Example

```
A_99_Z                          -- Identifier
\$%^&*()\                       -- Extended identifer
"+"                             -- Operator
IEEE.STD_LOGIC_1164."nand"      -- Selected name
RecordVariable.ElementName      -- Selected name
Vector(7)                       -- Indexed name
Matrix(I, J, K)                 -- Indexed name
Vector(23 downto 16)            -- Slice name
Vector(J to K)                  -- Slice name
Clock'EVENT                     -- Attribute name
```

See Also

Attribute Name, Operator, Expression, Range, Array, Record

New

Used to dynamically allocate memory for an access type. Given a data type, **new** allocates storage for a value of that type, and returns a pointer to that value. Optionally, a qualified expression can be used to initialize the newly allocated value.

Syntax

{either}

```
new DataType
new QualifiedExpression
```

Where

See Expression

Synthesis

Not synthesizable.

Tips

Use the built in procedure DEALLOCATE(Ptr) to release memory allocated with **new**.

Example

```
Link := new Std_logic_vector(7 downto 0);
NewPointer := new Word'(Link, "10000000");
```

See Also

Access, Type, Qualified Expression, Null

A sequential statement which jumps back to the top of a loop. In the case of a **for loop**, the loop parameter takes the next value in its range.

Syntax

```
[Label:] next [LoopLabel] [when Condition];
```

Where

```
loop-<HERE>-end loop
```

See Sequential Statement

Rules

The **next** must be inside a loop with the given LoopLabel, or inside any loop if there is no LoopLabel.

Example

```
L1: loop
  L2: for I in A'RANGE loop
    next when I = N;     -- Jump to next iteration of inner loop L2
    if A(I) = 'U' then
      next L1;           -- Jump to top of outer loop L1
    end if;
  end loop L2;
  ...
end loop L1;
```

See Also

Exit, For Loop, While Loop, Loop

Null

A sequential statement that does nothing, normally used within an *if* or *case* statement as an explicit way to take no action under certain conditions. Also, the value assigned to a guarded signal to mean disconnection, and the value assigned to an access variable by the procedure DEALLOCATE(Ptr).

Syntax

```
[Label:] null;              {sequential statement}
null                        {value}
```

Where

See Sequential Statement and Expression respectively.

Rules

The null statement is not mandatory - it is possible to leave a blank instead.

Example

```
case Flag is
when TRUE =>
  Q := null;                -- null value
when FALSE =>
  null;                     -- null statement
end case;
```

See Also

Case, If, Access, Disconnect

An integer or real mathematical number. The default number base is decimal. For based integers with exponents, the BasedInteger value is multiplied by the Base raised to the power of the Exponent.

Syntax

{either}
Integer[.*Integer*][*Exponent*]
Base#*BasedInteger*[.*BasedInteger*]#[*Exponent*]

Base = *Integer* {decimal number between 2 and 16}
BasedInteger = {hex digits 0-9, A-F and underscores}
Exponent = E+/-*Integer*
Integer = {decimal digits 0-9 and underscores}

Where

See Expression

Rules

- A number must not contain embedded spaces, nor can it contain adjacent underscores.
- A number with no radix point is an integer; the exponent of an integer must not be negative.

Example

```
30 = 3E1 = 16#1E# = 2#11_11#e1
30.0 = 300.0e-1 = 16#1E.0# = 2#11.11#E+3
```

See Also

Integer, Floating, String

Numeric_Std

Numeric_std is a new IEEE standard package which defines arithmetic operations on arrays of Std_logic. The intent is that this package should be supported by all synthesis tools. The package defines two new types to represent numbers:

```
type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
type SIGNED   is array (NATURAL range <>) of STD_LOGIC;
```

Operators overloaded on combinations of the types UNSIGNED and NATURAL, and on combinations of the types SIGNED and INTEGER:

```
"+" "-" "*" "/" "rem" "mod" ">" ">=" "<" "<=" "=" "/="
```

Shift operators overloaded on types UNSIGNED and SIGNED:

```
"sll" "srl" "rol" "ror"
```

Operators overloaded on type SIGNED only:

```
"abs" "-" {signs}
```

Logical operators overloaded on types UNSIGNED and SIGNED:

```
"not" "and" "or" "nand" "nor" "xor" "xnor"
```

Functions to do signed and unsigned extension:

```
function RESIZE (ARG: SIGNED; NEW_SIZE: NATURAL)
               return SIGNED;
function RESIZE (ARG: UNSIGNED; NEW_SIZE: NATURAL)
               return UNSIGNED;
```

Type conversion functions:

```
function TO_INTEGER (ARG: UNSIGNED)   return NATURAL;
function TO_INTEGER (ARG: SIGNED)     return INTEGER;
function TO_UNSIGNED(ARG, SIZE: NATURAL)
                                      return UNSIGNED;
function TO_SIGNED  (ARG: INTEGER; SIZE: NATURAL)
                                      return SIGNED;
```

Functions which match '-' with any value, but match 'U', 'X', 'W' and 'Z' with nothing except '-'.

```
function STD_MATCH (L, R: STD_ULOGIC) return BOOLEAN;
function STD_MATCH (L, R: UNSIGNED)   return BOOLEAN;
function STD_MATCH (L, R: SIGNED)     return BOOLEAN;
function STD_MATCH (L, R: STD_LOGIC_VECTOR)
                                      return BOOLEAN;
function STD_MATCH (L, R: STD_ULOGIC_VECTOR)
                                      return BOOLEAN;
```

Functions which convert '0' and 'L' to '0', '1' and 'H' to '1', and everything else to XMAP

```
function TO_01 (S: UNSIGNED; XMAP: STD_LOGIC := '0')
                                      return UNSIGNED;
function TO_01 (S: SIGNED;   XMAP: STD_LOGIC := '0')
                                      return SIGNED;
```

See Also

Standard, Std_logic_1164

Operator

An operator is a logical or mathematical function which takes one or two values and produces a single result. Operators are built into the syntax of the language, i.e. you cannot define new operators. Operators can be redefined (overloaded in VHDL jargon) for any types by writing new functions.

Syntax

{2 operands: Expression Operator Expression}

```
+  -  *  /  mod  rem  **
=  /=  <  <=  >  >=
and  or  xor  nand  nor  xnor
sll  srl  sla  sra  rol  ror
&
```

{1 operand: Operator Expression}

```
+  -  abs  not
```

Where

```
function "<HERE>"
```

See Expression

Rules

- / on integers truncates toward 0.
- **mod** and **rem** give the remainder on division. A **rem** B has the sign of A, A **mod** B has the sign of B.
- A **sla** B replicates the rightmost bit of A, A **sra** B replicates the leftmost bit of A.
- A & B yields a vector whose length is the length of A + the length of B, with the content of A on the left and B on the right.

Gotchas!

- The default definitions of `>` `>=` `<` `<=` give unexpected results for vectors of different lengths, e.g. "000" > "00" !
- When the "+" operator is overloaded on type Std_logic_vector, it is usual to make the width of the result equal to that of the widest operand, with the effect that the result is truncated, e.g. "1" + "1" = "0" and "111" + "1" = "000".

Synthesis

- The operators `+ - = /= < <= > >=` are synthesizable as adders, subtractors and comparators.
- The operators / mod rem ** are not synthesizable in general, but may be synthesized when used to do masks and shifts or in static expressions (e.g. A / 2 means shift right).

See Also

Expression, Function

Package

A package contains common definitions that can be shared across a VHDL design or even several designs. A package is split into a declaration and a body. The package declaration defines the external interface to the package, the package body typically contains the bodies of any functions or procedures defined in the package declaration.

Syntax

```
{declaration}
package PackageName is
  Declarations...
end [package] [PackageName];

{body}
package body PackageName is
  Declarations...
end [package body] [PackageName];
```

Where

See (VHDL) File

Tips

Common, shared declarations of types, subtypes, constants, procedures, functions and components are best put in a package.

Gotchas!

- Where a function or procedure is placed in a package, the declaration and body must conform, i.e. the parameters must be identical between the two.
- Only definitions placed in the package declaration are visible outside the package.

Example

(See Function)

```
library IEEE;
use IEEE.Std_logic_1164.all;

package UTILITIES is
  -- Declarations...
  subtype Byte is Std_logic_vector(7 downto 0);
  function PARITY (V: Byte) return Std_logic;
end UTILITIES;
```

```
package body UTILITIES is
  -- Bodies...
  function PARITY (V: Byte) return Std_logic is
    variable B: Std_logic := '0';
  begin
    for I in V'RANGE loop
      B := B xor V(I);
    end loop;
    return B;
  end PARITY;
end UTILITIES;
```

See Also

(VHDL) File, Use, Function, Procedure, Declaration, Type, Component

Physical

A physical type represents an integer value together with a physical unit.

Syntax

```
type NewName is range Range      {static integer range}
  units
    PrimaryUnitName;
    SecondaryUnitName = PhysicalLiteral;
    SecondaryUnitName = PhysicalLiteral;
    ... ;
  end units [NewName];

PhysicalLiteral = Number UnitName
```

Where

See Declaration

Synthesis

Not synthesizable.

Tips

The only commonly used physical type is TIME. You are unlikely to need to define new physical types.

Example

```
type Distance is range 0 to INTEGER'HIGH
  units
    micron;
    millimetre = 1000 micron;
    centimetre = 10 millimetre;
    metre = 100 centimetre;
  end units;
```

See Also

Type, Integer, Range

A port represents a pin or a related group of pins on a hardware component, and is defined in an entity. Technically, a port is a signal.

Syntax

```
port (PortName, ... : [Mode] DataType [:= Expression];
      ...);

Mode = {either} in out inout buffer linkage
```

Where

```
entity-is-generic(-);-<HERE>-begin-end
component-generic(-);-<HERE>-end
block-generic map(-);-<HERE>-port map-begin-end
```

Rules

- **In** ports can only be read, **out** ports can only be assigned. **Inout** ports are bidirectional. **Buffer** ports are outputs.
- The *Expression* gives the default value for the port, and must be static. An input port with no default value must appear in the corresponding port map.

Gotchas!

- You cannot read the value of an **out** port within an architecture.
- A **buffer** port cannot be connected to an **in** or **out** port of the design entity containing the instantiation; it can be connected to a **buffer** of that design entity.

Tips

- Do not put ports in test bench entities.
- Outputs can be **out** ports or **buffer** ports. **Buffer** must be used when the value of the port is to be read inside the architecture, and **out** must be used when there is more than one driver for the signal.
- **Linkage** ports cannot be read or assigned, so are not typically used.

Example

```
port (Clock, Reset: in Std_logic;
      Q: buffer Std_logic_vector(7 downto 0);
      Status: out Std_logic_vector);
```

See Also

Entity, Component, Port Map, Block, Generic

Port Map

A port map is typically used to define the interconnection between instances in a structural description (or netlist). A **port map** maps signals in an architecture to ports on an instance within that architecture. Port maps can also appear in a configuration or a block.

Syntax

```
port map ([Formal =>] Actual, ...)

Formal = {either} Name FunctionCall
Actual = {either} Name FunctionCall open
```

Where

```
Label:ComponentName generic map(-)<HERE>;
for-use-generic map(-)<HERE>;
block-port(-);<HERE>;-begin-end
```

Rules

- The two forms of syntax (ordered list or explicitly named ports) can be mixed, but the ordered list must come before the named ports.
- Within an instance, the formals are ports on the component or entity being instanced, the actuals are signals visible in the architecture containing the instance.
- Within a configuration, the formals are ports on the entity, the actuals are ports on the component.
- If the actual is a conversion function, this is called implicitly as values are passed in.
- If the formal is a conversion function, this is called implicitly as values are passed out.

Tips

Use the port names rather than order to improve readability and reduce the risk of making connection errors.

Example

```
component COUNTER
  port (CLK, RESET: in Std_logic;
        UpDown: in Std_logic := '0';   -- default value
        Q: out Std_logic_vector(3 downto 0));
end component;
...
-- Positional association...
G1: COUNTER port map (Clk32MHz, RST, open, Count);

-- Named association (order doesn't matter)...
G2: COUNTER port map (  RESET => RST,
                        CLK => Clk32MHz,
                        Q(3) => Q2MHz,
                        Q(2) => open,  -- unconnected
                        Q(1 downto 0) => Cnt2,
                        UpDown => open);
```

See Also

Port, Instantiation, Component, Block, Generic Map

Procedure

Used to group together executable, sequential statements. A procedure has a name and a set of parameters. The values of the parameters are passed in or out when the procedure is called. When defined in a package, the procedure must be split into a declaration and a body.

Syntax

```
{declaration}
procedure ProcedureName [(ParameterDeclaration; ...)];

{body}
procedure ProcedureName [(ParameterDeclaration; ...)] is
  Declarations...
begin
  SequentialStatements...
end [procedure] [ProcedureName];

ParameterDeclaration = {either}
constant ConstantName,...: [in]  DataType [:=Expression]
signal   SignalName,  ...: [Mode]DataType [:=Expression]
variable VariableName,...: [Mode]DataType [:=Expression]
file FileName, ... : DataType

Mode = {either} in out inout
```

Where

See Declaration

A Procedure Body is not allowed in a Package Declaration.

Rules

- *Mode* defaults to **in**. Parameters of mode **in** default to constant. Parameters of mode **out** or **inout** default to variable.
- The *Expression* gives the default value of the parameter. A parameter with no default value must be given a value in the procedure call.
- A procedure containing a signal assignment (other than to a parameter of the procedure) must be declared inside a process.

Gotchas!

- Variables defined inside a procedure are initialized each time the procedure is called.
- A procedure containing assignments to signals (other than parameters) must be defined in a process.
- The procedure declaration and body must conform, i.e. the parameters must be identical between the two.
- The procedure declaration ends with a ";", whereas the procedure body has **is** at the corresponding point.

Synthesis

Procedures are synthesizable, provided that they do not detect clock edges or contain **wait** statements, i.e. they must not infer registers or states.

Tips

Parameters may be unconstrained arrays; you can use array attributes (e.g. 'RANGE) to find their bounds.

Example

```
procedure ASSIGN (signal Clock: in Std_logic;
                  Values: Std_logic_vector;
                  signal X, Y: out Std_logic;
                  variable V, W: out Std_logic;
                  PAUSE: TIME := 10 NS) is
begin
  if Clock'EVENT and Clock = '1' then
    X <= Values(0);
    Y <= Values(1);
    V := Values(2);
    W := Values(3);
    wait for PAUSE;
  end if;
end ASSIGN;
...
-- Procedure call...
ASSIGN (Clock, "0101", S1, S2, V1, V2);
```

See Also

Procedure Call, Function, Package, Return

Procedure Call

A sequential or concurrent statement which causes a procedure to be executed. The values of any parameters are passed in or out of the procedure, depending on their *Mode*.

Syntax

```
[Label:] ProcedureName [ ( [Formal =>] Actual, ... ) ]

Formal = {either} Name FunctionCall
Actual = Expression
```

Where

```
entity-begin-<HERE>-end          {passive procedure call}
architecture-begin-<HERE>-end
block-begin-<HERE>-end
generate-begin-<HERE>-end
```

See Sequential Statement

Rules

- Sequential statements can be labelled in VHDL'93, but not in VHDL'87.
- The two forms of syntax (ordered list or explicitly named parameters) can be mixed, but the ordered list must come before the named parameters.

Synthesis

- A procedure call is synthesized by substituting the logic represented by the procedure in place of the call. Synthesis does not treat procedures as shareable resources.
- A procedure call in a clocked process may result in registers being inferred from signal assignments within the procedure.

Tips

Use the parameter names rather than order to improve readability and reduce the risk of making errors.

Example

```
procedure Write (L:          inout Line;
                 Value:      in Std_logic_vector;
                 Justified: in Side  := Right;
                 Field:      in Width := 0);
...
Write (Buff, A, Left, 8);
Write (Buff, C);
Write (Justified => Left, Field => 12,
       L => Buff, Value => D);
```

See Also

Procedure, Function Call

Process

A concurrent statement which describes behaviour. A process is itself a concurrent statement, but it contains sequential statements that execute in series from top to bottom. A process executes at times controlled by the sensitivity list or by **wait** statements.

Syntax

```
[Label:] [postponed] process [ (SensitivityList) ] [is]
  Declarations...
begin
  SequentialStatements...
end [postponed] process [Label];

SensitivityList = SignalName, ...
```

Where

```
entity-begin-<HERE>-end          {passive process}
architecture-begin-<HERE>-end
block-begin-<HERE>-end
generate-begin-<HERE>-end
```

Rules

- A process must contain either a sensitivity list or **wait** statements, but not both.
- Every process executes once during initialization, before simulation starts.
- A **postponed** process is not executed until the final simulation cycle of a particular simulation time, and thus sees the stable values of signals and variables.

Gotchas!

A process with neither a sensitivity list nor a **wait** will loop forever.

Synthesis

Processes are one of the most useful VHDL statements for synthesis, yet many processes are unsynthesizable. For best results, code should be restricted to the following process templates:

```
process (Inputs)            -- All inputs in sensitivity list
begin
  ...                       -- Outputs assigned for all input conditions
  ...                       -- No feedback
end process;                -- Gives pure combinational logic
```

```
process (Inputs)          -- All inputs in sensitivity list
begin
  if Enable = '1' then
    ...                   -- Latched actions
  end if;
end process;              -- Gives transparent latches + logic

process (Clock)           -- Clock only in sensitivity list
begin
  if Rising_edge(Clock) then -- Test clock edge only
    ...                   -- Synchronous actions
  end if;
end process;              -- Gives flipflops + logic

process (Clock, Reset)    -- Clock and reset only in sensitivity list
begin
  if Reset = '0' then     -- Test active level of asynchronous reset
    ...                   -- Asynchronous actions
  elsif Rising_edge(Clock) then -- Test clock edge only
    ...                   -- Synchronous actions
  end if;
end process;              -- Gives flipflops + logic

process                   -- No sensitivity list
begin
  wait until Rising_edge(Clock);
  ...                     -- Synchronous actions
end process;              -- Gives flipflops + logic
```

Example

The following example shows a Register Transfer Level process:

```
Counter: process (Reset, Clock)
begin
  if Reset = '0' then             -- Asynchronous reset
    Count <= (others => '0');
  elsif Rising_edge(Clock) then
    if Load = '0' then            -- Synchronous load
      Count <= Data;
    else
      Count <= Count + '1';
    end if;
  end if;
end process Counter;
```

The following example shows processes used to generate vectors in a test bench:

```
signal StopClock: Boolean;
signal Clk: Std_logic;
constant Period: Time := 10 NS;
subtype Int8 is Std_logic_vector(7 downto 0);
signal A, B: Int8;
type Operation is (Load, Store, Move, Halt);
signal Op: Operation;
...
ClockGenerator: process
begin
  while not StopClock loop
    Clk <= '0';
    wait for Period/2;
    Clk <= '1';
    wait for Period/2;
  end loop;
  wait;
end process ClockGenerator;

Stimulus: process
  type Table is array (Natural range <>) of Int8;
  constant Lookup: Table :=
    ("00000000", "00000001", "00000011", "00001000",
     "00001111", "10000000", "11111000", "11111111");
begin
  for L in 1 to 2 loop
    for I in Lookup'Range loop
      B <= Lookup(I);
      for J in Lookup'Range loop
        A <= Lookup(J);
        for K in Operation loop
          Op <= K;
          wait for Period;
        end loop;
      end loop;
    end loop;
    StopClock <= True;  -- Flag to stop clock generator process
  end loop;
  wait;
end process Stimulus;
```

See Also

Concurrent Statement, Sequential Statement, Wait

Used to define the type of an expression where otherwise the type would be ambiguous.

Syntax

```
{either}
TypeName'(Expression)
TypeName'Aggregate
```

Where

See Expression

Rules

The *Expression* or *Aggregate* must be compatible with the TypeName; a qualified expression is not a type conversion!

Tips

Use a qualified expression when the compiler gives an error indicating that the type of an expression is ambiguous. This can occur when calling overloaded functions and procedures (e.g. (1) and (2) below), and when constructing aggregates (e.g. (3) and (4) below).

Example

```
subtype T is STD_LOGIC_VECTOR(1 to 2);
...
if U > UNSIGNED'("10000000") then      -- (1)
  WRITE (L, STRING'("Hello"));         -- (2)
  V := (others => T'(others => '1'));  -- (3)
  case T'(A, B) is                     -- (4)
```

See Also

Aggregate, Expression, Type Conversion, TEXTIO

Range

A range specifies a range of values belonging to an integer, floating, physical or enumeration type. A static range is one whose bounds can be calculated during compilation or elaboration.

Syntax

{either}

Expression to *Expression*

Expression downto *Expression*

Name'RANGE {name of an array or array type}
Name'REVERSE_RANGE

DataType {enumeration or integer type}

Where

See Array, For Loop, Name, Aggregate, Case

Rules

The values of the *Expressions* must be consistent with the direction of the range. E.g. 0 downto 7 is a null range, of length 0.

Tips

- Any form of range can be used in a **for loop** or an index constraint.
- Use the 'RANGE form of range in preference to *Expression* to *Expression* where possible, as this often makes the code easier to maintain.

Example

```
subtype INT is INTEGER range 0 to 7;
subtype V1 is STD_LOGIC_VECTOR(INT);
subtype V2 is INTEGER range V1'REVERSE_RANGE;
...
for I in V2 range 3 downto 0 loop
  ...
```

See Also

Data Type, Subtype, Integer, Floating, Attribute Name

Record

A data type that represents a set of values of different types. Used to define data structures. Not typically used to describe hardware, but useful for high level modelling.

Syntax

```
type NewName is
  record
    ElementName, ... : DataType;
    ... ;
  end record [NewName];
```

Where

See Declaration

Gotchas!

Not all synthesis tools support record types.

Synthesis

- Values of a record types are synthesized as a bundle of wires, structured according to the elements of the record.
- Not all synthesis tools support records.

Tips

- The values within a record can be read or written using a selected name.
- Use record types together with access types to create dynamic data structures for high level modelling.

Example

```
type Floating is
  record
    Sign: Bit;
    Mantissa, Exponent: Integer;
  end record;
variable A, B: Floating;
...
A.Mantissa := A.Mantissa + B.Mantissa;
B := A;
```

See Also

Name, Type, Access, Array

Report

A sequential statement which writes out a text message to the simulator log.

Syntax

```
[Label:] report StringExpression [severity Expression];
```

Where

See Sequential Statement

Rules

The severity expression must be of type Severity_level, which has the values Note, Warning, Error, Failure. The default severity is Error.

Synthesis

Reports do not represent hardware. Synthesis tools ignore them or give a warning.

Example

```
report "Simulation finished" severity Note;
```

See Also

Assert, TEXTIO

This is a complete list of reserved identifiers in VHDL'87 and VHDL'93. These reserved identifers must not be used as user defined identifiers.

abs	file	of	sll
access	for	on	sra
after	function	open	srl
alias	generate	or	subtype
all	generic	others	then
and	group	out	to
architecture	guarded	package	transport
array	if	port	type
assert	impure	postponed	unaffected
attribute	in	procedure	units
begin	inertial	process	until
block	inout	pure	use
body	is	range	variable
buffer	label	record	wait
bus	library	register	when
case	linkage	reject	while
component	literal	rem	with
configuration	loop	report	xnor
constant	map	return	xor
disconnect	mod	rol	
downto	nand	ror	
else	new	select	
elsif	next	severity	
end	nor	signal	
entity	not	shared	
exit	null	sla	

Return

A sequential statement which causes execution to be returned from a procedure or function.

Syntax

```
[Label:] return;                     {in procedure}
[Label:] return Expression;          {in function}
```

Where

```
function-begin-<HERE>-end
procedure-begin-<HERE>-end
```

See Sequential Statement

Rules

A function must execute a return statement that returns a value consistent with the return type of the function.

Example

```
return A + B;
```

See Also

Function, Procedure, Expression

A concurrent statement which assigns one of several expressions to a signal, depending on the value of the expression at the top. Equivalent to a process containing a **case** statement.

Syntax

```
[Label:] with Expression select
  Target <= [Options]
    Expression [after TimeExpression] when Choices,
    Expression [after TimeExpression] when Choices,
    ... ;

Target = {either} SignalName Aggregate

Options = {either}
guarded
transport
reject TimeExpression inertial

Choices = Choice | ...

Choice = {either}
ConstantExpression
Range
others   {the last choice}
```

Where

```
architecture-begin-<HERE>-end
block-begin-<HERE>-end
generate-begin-<HERE>-end
```

Rules

- The reserved word **guarded** may only appear in a signal assignment within a guarded block. A guarded assignment only executes when the guard expression on the surrounding block is true.
- Every case of the *Expression* at the top must be covered once and only once by the choices.
- An *Expression* on the right hand side may be replaced by the reserved word **unaffected**.

Synthesis

Selected signal assignments are synthesized to combinational logic. The *Expressions* on the right hand side are multiplexed onto the *Target* signal.

Tips

- Conditional and selected signal assignments are a good way to describe combinational logic in Register Transfer Level descriptions.

Example

```
Mux: with S select
  F <= A when "000",
       B when "001",
       C when "010" | "011" | "100",
       D when others;
```

See Also

Signal Assignment, Conditional Assignment, Case, Block

Sequential statements execute in series, one after the other from top to bottom, so the order in which they are written is usually critical. The following are sequential statements:

- Wait
- Assert
- Report
- Signal Assignment
- Variable Assignment
- Procedure Call
- If
- Case
- For Loop
- While Loop
- Loop
- Next
- Exit
- Return
- Null

Where

```
process-begin-<HERE>-end
function-begin-<HERE>-end
procedure-begin-<HERE>-end
if-then-<HERE>-elsif-then-<HERE>-else-<HERE>-end
case-=>-<HERE>-when-=>-<HERE>-end
loop-<HERE>-end
```

See Also

Concurrent Statement

Shared Variable

Used to share information between processes. Intended for high level system modelling and for instrumenting code. The rules governing their use have not yet been standardised, so don't use them!

Syntax

```
shared variable VariableName, ... : DataType
                                          [:= Expression];
```

Where

See Declaration

Not allowed in Process, Function or Procedure

Gotchas!

Any non-trivial use of shared variables causes non-deterministic behaviour!

Synthesis

Shared variables cannot be synthesized.

See Also

Variable, Signal

Signal

A signal represents an electrical connection, wire or bus. Signals are used for communication between processes.

Syntax

```
signal SignalName, ... : DataType [Kind] [:=Expression];

Kind = {guarded signal, either} register bus
```

Where

```
package-<HERE>-end
entity-is-<HERE>-begin-end
architecture-is-<HERE>-begin-end
block-<HERE>-begin-end
generate-<HERE>-begin-end
```

See Declaration

Rules

- A signal can be assigned in more than one process only if it has a resolution function.
- A guarded signal can have individual drivers disconnected from the resolution function. A **register** with no drivers connected retains its previous value.
- The *Expression* gives the initial value of the signal at time zero.

Gotchas!

The initial value only initializes drivers within the scope of the signal declaration, so drivers for the same signal in other architectures are not initialized!

Synthesis

- The initial value is ignored for synthesis, so be careful! Resolution functions are generally ignored too.
- Guarded signals are not synthesizable in general.

Tips

Guarded signals (**register** and **bus**) are obscure and are generally to be avoided.

Example

```
signal A, B: Std_logic_vector(3 downto 0) := "ZZZZ";
```

See Also

Signal Assignment, Data Type, Disconnect, Block, Shared Variable

Signal Assignment

A sequential or concurrent statement which creates events on a signal.

Syntax

```
[Label:]                       {see Rules}
Target <= [Options] Expression [after TimeExpression],
                    Expression [after TimeExpression],
                    ... ;

Target = {either} SignalName Aggregate

Options = {either}
guarded                        {must be in a guarded block}
transport
reject TimeExpression inertial
```

Where

```
architecture-begin-<HERE>-end
block-begin-<HERE>-end
generate-begin-<HERE>-end
```

See Sequential Statement

Rules

- Sequential statements can be labelled in VHDL'93, but not in VHDL'87.
- The default delay mode (**inertial**) means that pulses shorter than the delay (or the **reject** period if specified) are ignored. **Transport** means that the assignment acts as a pure delay line.
- All delays are relative to the time when the assignment executes.
- A signal assignment with no delay or zero delay will cause an event after a delta delay, which means that the event happens only when all of the currently active processes have finished executing (i.e. after one simulation cycle).
- A process containing one or more assignments to a signal has a driver for that signal.
- For a signal with multiple drivers, the values of all the drivers are passed to the resolution function which calculates the value of the signal.
- Assigning the value **null** to a guarded signal disconnects the driver from the resolution function.
- A guarded assignment only executes when the guard expression at the top of the surrounding block is true.

Gotchas!

A signal assignment does not immediately change the value of the signal; there is always a delay of at least one delta.

Synthesis

- Delays are ignored for synthesis; use tool specific timing constraints instead.
- The *Expression* on the right hand side is synthesized as combinational logic.
- The *Target* is synthesized as a connection in a combinational processes, as a transparent latch when incompletely assigned in a combinational process, or as a register in a clocked process.

Tips

- Multiple events can be created by one signal assignment to define test vectors (see last example below), but it is better to use several simple signal assignments separated by **wait for** statements (see *Wait*).
- Simulation speed is dependent on the number of signal assignments executed. To speed up simulation, use fewer signals (which often means fewer processes and more variables), and use integer or enumeration types instead of arrays.

Example

```
A <= B;
A <= B nand C;
A <= B nand C after 0.2 NS;
(Cout, Sum) <= T'(A + B + Cin);
H <= "00", "01" after 10 NS, "10" after 20 NS;
```

See Also

Signal, Aggregate, Expression, Block, Conditional Assignment, Select, Disconnect, Variable Assignment

The package STANDARD in library STD is part of the VHDL standard.

VHDL Source Of Package

```
package STANDARD is
  type BOOLEAN is (FALSE, TRUE);
  type BIT is ('0', '1');
  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);
    --  VHDL'93 includes all 256 ASCII characters
  type SEVERITY_LEVEL is
      (NOTE, WARNING, ERROR, FAILURE);
  type INTEGER is range implementation_defined;
  type REAL is range implementation_defined;
  type TIME is range implementation_defined
    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;

  --  function that returns the current simulation time:
  function NOW return TIME; --(VHDL'87)
  subtype DELAY_LENGTH is TIME range 0 FS to TIME'HIGH;
  impure function NOW return DELAY_LENGTH;

  subtype NATURAL  is INTEGER range 0 to INTEGER'HIGH;
  subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;
  type STRING is array (POSITIVE range <>) of CHARACTER;
```

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

    type FILE_OPEN_KIND is
        (READ_MODE, WRITE_MODE, MODE);
    type FILE_OPEN_STATUS is
        (OPEN_OK, STATUS_ERROR, NAME_ERROR, MODE_ERROR);

    attribute FOREIGN: STRING;
  end STANDARD;
```

Tips

- The attribute FOREIGN can be attached to a procedure, function or architecture to indicate that it is defined in a language other than VHDL (usually C).
- Generally it is better to use the types Std_logic and Std_logic_vector rather than Bit and Bit_vector.

See Also

TEXTIO, Std_logic_1164, Package

Std_Logic_1164

Std_logic_1164 is a VHDL package in the library IEEE. It is the IEEE standard for describing digital logic values in VHDL. It defines type Std_logic to represent a single bit of digital information, and Std_logic_vector to represent a bus. It also contains VHDL functions for these types to resolve tristate bus conflicts, functions to define logical operators, and conversions functions to and from other standard data types. Significantly, it does not contain any arithmetic or comparison operators, or any conversion functions to and from integer types. These are usually found in packages from synthesis tool vendors, or in the new IEEE 1076.3 standard package NUMERIC_STD.

VHDL Source Of Package

```
package Std_logic_1164 is
  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;

  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';

  function "and"  (L, R: Std_ulogic) return UX01;
  function "nand" (L, R: Std_ulogic) return UX01;
  function "or"   (L, R: Std_ulogic) return UX01;
  function "nor"  (L, R: Std_ulogic) return UX01;
  function "xor"  (L, R: Std_ulogic) return UX01;
  function "xnor" (L, R: Std_ulogic) return UX01;
  function "not"  (L:    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;
    function "nand" (L, R: Std_logic_vector)
                    return Std_logic_vector;
    function "nand" (L, R: Std_ulogic_vector)
                    return Std_ulogic_vector;
    function "or"   (L, R: Std_logic_vector)
                    return Std_logic_vector;
    function "or"   (L, R: Std_ulogic_vector)
                    return Std_ulogic_vector;
    function "nor"  (L, R: Std_logic_vector)
                    return Std_logic_vector;
    function "nor"  (L, R: Std_ulogic_vector)
                    return Std_ulogic_vector;
    function "xor"  (L, R: Std_logic_vector)
                    return Std_logic_vector;
    function "xor"  (L, R: Std_ulogic_vector)
                    return Std_ulogic_vector;
    function "xnor" (L, R: Std_logic_vector)
                    return Std_logic_vector;
    function "xnor" (L, R: Std_ulogic_vector)
                    return Std_ulogic_vector;
    function "not"  (L:    Std_logic_vector)
                    return Std_logic_vector;
    function "not"  (L:    Std_ulogic_vector)
                    return Std_ulogic_vector;

    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;
    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;

    function Rising_edge  (signal S: Std_ulogic)
                        return BOOLEAN;
    function Falling_edge (signal S: Std_ulogic)
                        return BOOLEAN;

    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;
  end Std_logic_1164;
```

Gotchas!

- Rising_edge and Falling_edge are the best way to detect clock edges, but not all synthesis tools support these functions! The alternative to Rising_edge(Clock) is to use the expression:
  `Clock'EVENT and Clock = '1'`

- The don't care value '-' is intended to be used as an output don't care, not an input don't care. A comparison like A = '-' will be false unless A is literally '-' !

Tips

The IEEE standard data types are Std_logic and Std_logic_vector, which are resolved subtypes, so can be used for signals with more than one driver (i.e. tristate busses). The types Std_ulogic and Std_ulogic_vector are unresolved types, so signals with these types can have only one driver. It is a good idea to use types Std_ulogic and Std_ulogic_vector for signals that are intended to have just one driver (i.e. most signals in a design!), since if two outputs are inadvertently wired together, this will be immediately reported as an error by the simulator. However, Std_ulogic and Std_ulogic_vector are not the standard types, so their use might not be supported by some tools.

See Also

Standard, Numeric_std, Package

String

A string is a value for a one-dimensional array of characters. A string of '0's and '1's may be written in octal or hexadecimal instead of binary. String is also the name of a type defined in package Standard.

Syntax

```
"{any string of printable characters}"
B"BitValue"                     {binary}
O"BitValue"                     {octal}
X"BitValue"                     {hexadecimal}

BitValue = {hex digits 0-9, A-F and underscores}
```

Where

See Expression

Gotchas!

- A value of type Std_logic_vector may be written as a string with an explicit number base in VHDL'93, but must be written without a number base in VHDL'87.
- Underscores are only ignored if the string has an explicit number base.
- Other values such as 'X' or 'Z' are not allowed if the string has an explicit number base.
- Strings are case sensitive, e.g. "xxxx" is not the same as "XXXX".

Example

```
"Hello"
"0000XXXX"
B"0000_0101"
X"FFFF"
```

See Also

Number, Enumeration, Standard

Subtype

A subtype declaration is used to give an explicit name to a subtype, which is itself a type plus a constraint. The constraint is used to restrict the values that a data object can take during simulation or synthesis. An anonymous subtype occurs when a *DataType* with a constraint is used directly without declaring a named subtype.

Syntax

```
subtype NewName is DataType;
```

Where

See Declaration

Rules

- A subtype is compatible with its base type, and shares the same operations as its base type.
- An array type can only be constrained once, whereas with an integer or enumeration type you can create subtypes of subtypes ad infinitum.

Example

```
subtype Std_logic is Resolved Std_ulogic;
subtype MyBit is Std_logic range '0' to '1';
subtype ShortVector is Std_logic_vector(1 downto 0);
```

See Also

Type, Data Type, Range

Textio

TEXTIO is a VHDL package which allows the reading and writing of ASCII text files from VHDL. TEXTIO is part of the IEEE 1076 standard, and is in the library STD.

VHDL Source Of Package

```
package TEXTIO is
  -- Type definitions for Text I/O
  type LINE is access STRING;
  -- a LINE is a pointer to a STRING value
  type TEXT is file of STRING;
  -- a file of variable-length ASCII records
  type SIDE is (RIGHT, LEFT);
  -- for justifying output data within fields
  subtype WIDTH is NATURAL;
  -- for specifying widths of output fields

  -- Standard Text Files (VHDL'87 version is similar)
  file INPUT:  TEXT open READ_MODE  is "STD_INPUT";
  file OUTPUT: TEXT open WRITE_MODE is "STD_OUTPUT";

  -- Input Routines for Standard Types
  procedure READLINE (variable f: in TEXT;
                      L: inout LINE);
  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);
```

```
    --  Output Routines for Standard Types
    procedure WRITELINE (F: out TEXT; L: inout LINE);
    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);

  -- function ENDFILE(F: in TEXT) return BOOLEAN;
  --  ENDFILE is implicitly declared for all file types
  end TEXTIO;
```

Rules

- READ skips leading white space except when reading types CHARACTER and STRING.
- The FIELD parameter to WRITE gives the minimum field width.

Gotchas!

- The READ and WRITE procedures are only defined for types in package STANDARD, not for user defined types.
- The function ENDLINE (as defined in the VHDL'87 standard) is not legal VHDL, so cannot be used!
- To WRITE a text string, a qualified expression must be used, e.g. `WRITE (L, STRING'("Hello World!"));`

Tips

- For a line that is being read, L'LENGTH = 0 when the end of the line has been reached.
- In VHDL'95, the attributes 'IMAGE and 'VALUE are useful for writing and reading non-standard data types.

Example

```
--  Writing a text file
signal A, B, G: Bit_vector(3 downto 0);
...
Monitor: process
  use STD.TEXTIO.all;
  file F: TEXT is out "test.txt";               --VHDL'87
  file F: TEXT open WRITE_MODE is "test.txt";
  variable L: LINE;
begin
  --  Strobe the signals...
  wait until Rising_edge(Clock);
  wait for Settling_time;

  WRITE (L, NOW, Left, 10); --  NOW = current simulation time
  WRITE (L, A,   Right, 5);
  WRITE (L, B,   Right, 5);
  WRITE (L, G,   Right, 5);

  WRITELINE (F, L);
end process;

--  Reading a text file
signal A, B: BIT_VECTOR(3 downto 0);
...
Stimulus: process
  use STD.TEXTIO.all;
  file F: TEXT is in "vectors.txt";             --VHDL'87
  file F: TEXT open READ_MODE is "vectors.txt";
  variable L: LINE;
  variable TimeWhen: TIME;
  variable Avalue, Bvalue: BIT_VECTOR(3 downto 0);
begin
  while not ENDFILE(F) loop
    READLINE (F, L);
    READ (L, TimeWhen);
    READ (L, Avalue);
    READ (L, Bvalue);
    wait for TimeWhen-NOW; --  Wait until an absolute time
    A <= Avalue;
    B <= Bvalue;
  end loop;
  wait;
end process;
```

See Also

File, Report, Standard, Attribute Name

Type

All signals, variables, constants (i.e. objects) and expressions have a type. The type defines the set of values that the object or expression can have. A type also determines the set of operations (operators, functions and procedures) which can be performed on that object or expression.

There are eight kinds of type:

- Access
- Array
- Enumeration
- File
- Floating
- Integer
- Physical
- Record

Rules

- A value of one type cannot be assigned to an object of a different type.
- The initial value of an object is the leftmost value of the type, i.e. the value on the left of the base enumeration, floating, integer or physical type.

Tips

Types are often best defined in packages, which contain common, shared definitions.

See Also

Subtype, Data Type, Package

Type Conversion

Used to convert between two closely related types. Such type conversions are implicitly defined by the language. Closely related types are either numbers (i.e. integer or floating point types), or two array types with similar definitions.

Syntax

```
TypeName (Expression)
```

Where

See Expression

Rules

TypeName must be the name of an integer, floating or array type.

Gotchas!

Type conversions are only defined implicitly for closely related types. For other types, you must write explicit conversion functions.

Example

```
signal I: Integer;
signal R: Real;
...
R <= Real(I) * 2.0;
...
type Std_logic_vector is
  array (Natural range <>) of Std_logic;
type Unsigned is
  array (Natural range <>) of Std_logic;
-- Std_logic_vector and Unsigned are closely related
signal S: Std_logic_vector(7 downto 0);
signal U: Unsigned(7 downto 0);
...
S <= Std_logic_vector(U);
U <= Unsigned(S);
```

See Also

Function, Type, Integer, Floating, Array, Qualified Expression

Makes names defined in a library or package directly visible within another region of VHDL code. Typically written either at the top of an entity, giving access to common definitions from a package, or at the top of a configuration, giving access to the entities and architectures from a library.

Syntax

```
use SelectedName, ... ;

SelectedName = {typically one of}
LibraryName.PackageName.ItemName
LibraryName.PackageName.all
LibraryName.ItemName
LibraryName.all
```

Where

See Declaration, (VHDL) File, Configuration

Rules

If two **use**s attempt to make the same name visible from different places, then neither is visible. Similarly, a name made visible by a **use** will be hidden by a local definition of the same name.

Gotchas!

use Lib.Pack.TypeName; does not make enumeration literals, operators or implicit declarations visible!

Tips

If two **use**s cancel each other out, the hidden item can be referenced using its selected name (e.g. Lib.Pack.Item).

Example

```
use IEEE.Std_logic_1164.all;     --  The contents of the package
use WORK.Arith_Ops;              --  The package name only
use CMOS_TECH.all;               --  All the entities in the library
```

See Also

Library, (VHDL) File, Package, Name

Variable

A variable stores a value within a process.

Syntax

```
variable VariableName, ... : DataType [:= Expression];
```

Where

```
process-<HERE>-begin-end
function-is-<HERE>-begin-end
procedure-is-<HERE>-begin-end
```

See Declaration

Rules

The expression gives the initial value of the variable. A variable declared in a process is initialized once during elaboration. A variable declared in a function or procedure is re-initialized on each call.

Gotchas!

A variable cannot be declared in an architecture; you must use a **shared** variable or signal to communicate between processes.

Synthesis

- A variable represents either a wire or a register, depending on whether the value is stored between clock cycles.
- Do not use a variable to store a value between executions of an unclocked process, because simulation and synthesis will inevitably give different results!
- The initial value is ignored for synthesis, so be careful!

Example

```
variable V, W: INTEGER range 0 to 7 := 7;
```

See Also

Variable Assignment, Shared Variable, Signal

A sequential statement which changes the value stored in a variable. A variable assignment has no delay.

Syntax

```
[Label:] Target := Expression;

Target = {either} VariableName Aggregate
```

Where

See Sequential Statement

Synthesis

- The *Expression* on the right hand side is synthesized as combinational logic.
- The *Target* is synthesized to a wire, a latch or a flipflop depending on whether the value is stored between clock cycles:

```
process
  variable V, W: Std_logic;
begin
  wait until Clock = '1';
  V := A nand W;          -- a nand gate
  V := V nor B;           -- a nor gate
  W := D;                 -- a flipflop
  S <= V;                 -- a flipflop
end process;
```

Example

```
V := V + 1;
(V, W) := X;
```

See Also

Variable, Shared Variable, Signal Assignment, Expression

Vhdl 93

The new features of VHDL'93 are listed below; topic references are given in brackets:

- Consistent statement bracketing (Entity, Architecture, Configuration, Package, Component, Procedure, Function, Process, Generate, If, Case, Loop, Record)
- Direct instantiation of entities and configurations, avoiding components (Instantiation)
- Opening, closing and appending to files (File)
- Shared variables (Shared variable)
- New attributes, including 'IMAGE and 'PATH_NAME (Attribute name)
- Report statement for writing messages (Report)
- New shift, rotate and xnor operators (Operator)
- Generalized hex and octal bit string literals (String)
- Unaffected value in conditional signal assignments (Conditional Assignment, Select)
- Variable pulse rejection for inertial delay (Signal assignment)
- Postponed processes which execute in the last delta of a simulation time (Process)
- Pure and impure functions (Function)
- Declarations within generate statements (Generate)
- Generalized aliasing, such that anything can be aliased (Alias)
- Groups, typically used to pass information to synthesis tools (Group)
- Labels on sequential statements (Sequential Statement)
- Extended identifiers, allowing any printable character in a name (Name)

A sequential statement which waits for an event on a signal in the sensitivity list, for a *Condition* to become true, or for a timeout.

Syntax

```
[Label:] wait [on SensitivityList] [until Condition]
              [for TimeExpression];

SensitivityList = SignalName, ...
```

Where

See Sequential Statement

Rules

- When an event occurs on a signal in the sensitivity list, the *Condition* is checked. If True, the process resumes. In any case, the process resumes after the *TimeExpression* has elapsed.
- If the **on** part is omitted, a sensitivity list is built from the signals within the *Condition*.

Gotchas!

Wait until is edge triggered; the *Condition* is only tested when an event occurs on a signal in the wait statement. Thus, **`wait until Now = 1 US;`** would wait forever!

Synthesis

- **Wait for**, **wait on** and **wait;** are not synthesizable.
- **Wait until** is synthesizable only when used to synchronize a process to a clock edge, e.g. **wait until Clock = '1';** (see Process).

Tips

The **wait for** and **wait;** statements are particularly useful for defining test vectors, as shown in the example below. The **wait;** at the end of the process is necessary to stop the process looping.

Example

(See Process)

```
-- Generating Test Vectors:
```

```
constant Period: TIME := 25 NS;
...
Stimulus: process
begin
  A <= "0000";
  B <= "0000";
  wait for Period;
  A <= "1111";
  wait for Period;
  B <= "1111";
  wait for Period;
  wait;
end process;
```

See Also

Process

A sequential statement. The statements inside the while loop are executed repeatedly while the condition is True.

Syntax

```
[LoopLabel:] while Condition loop
  SequentialStatements...
end loop [LoopLabel];
```

Where

See Sequential Statement

Rules

The *Condition* is tested at the start of the loop and once after each iteration, but not during the execution of the *Statements*.

Synthesis

Not generally synthesizable. Some tools do allow **while loops** containing **wait** statements to describe implicit finite state machines, but this is not recommended practice.

Example

```
while Going loop
  Count := Count + 1;
  wait until Clock = '1';
end loop;
```

(See Access, Process, TEXTIO)

See Also

For Loop, Loop, Exit, Next

Index

Index

Index

Index

Index

Index

Index