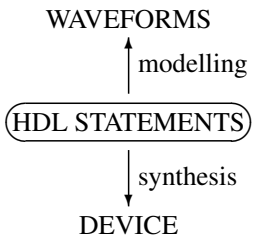
2 Introduction to VHDL

Any Hardware Description Language (HDL) aims at describing variety of aspects of digital devices like their **structure** and **functional and time behaviour** in order to perform either modelling, simulation and testing of a device, or to synthesize it in a selected technology.

Typically, we can assume that any HDL description can be converted into time **waveforms** specifying the time behaviour of a digital device, or into a structural or physical domain in a specified technology, as illustrated in the following diagram:



For example a statement

```
clk <= NOT clk AFTER 50 ns
```

can describe generation of a signal, or a waveform clk with the period of 100 ns, whereas, a similar statement

```
y <= NOT x
```

can describe an inverter in any underlying technology.

VHDL is one of the most popular Hardware Description Languages and according to the above introductory remarks can be used to

- model the functional and the time behaviour of digital systems (**modelling, simulation and testing**)
- describe or infer the structure of digital hardware, that is, give the circuit **specification** and/or describe the way in which the circuit can be build **synthesis**)

The syntax of VHDL was influenced by the ADA language and was initially developed by the US DoD (1981). The first IEEE standard was published in 1987 (VHDL–87). The revised standard is called VHDL 1076-1993, and its current revision can be found in [1]. Full description of VHDL is available in [2].
From the language specification it transpires that the structure of VHDL is rather complex, not only in order to make possible the description of the two relatively different sides of digital systems, namely, their **structure** and **behaviour**, but also to provide very flexible language mechanisms. One of the aspects worth noticing is that components of digital devices operate **concurrently**, therefore, the VHDL is inherently a parallel or concurrent language.
Despite of its complexity, it is possible to specify a subset used for **synthesis** which is significantly simpler. It is because many essential language constructs, like file operations, complex data structures, explicit time references, etc., do not directly represent any sensible hardware. A formal specification of the possible synthesis subset is given in the IEEE standards [3, 4]. See also [5].

2.1 Basic terminology

Component

- A component is a central concept in description of digital hardware using VHDL and is used to hierarchically represent digital circuitry from a simple gate to a complex processing system.
- A component representing digital hardware can be thought of as a box in which we specify the **input-output ports** and the component's structure or behaviour.

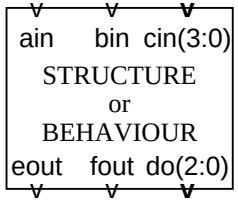


Figure 1: A component

- Ports can represent single wires (or signals), e.g. ain, bin, eout, fout, or a group of wires (or signals), that is, buses (vectors of signals) e.g. cin(3:0) and do(2:0). Ports can be, in general, of the input, output and bi-directional (or input-output) type.
- One component can be **instantiated** in another using various language mechanisms.
- Description of a component consists of the **entity** and **architecture**

Entity

Typically, the **entity** is the declaration of the component's input-output **ports** and its **name**.

Example: an entity for a 1-bit adder

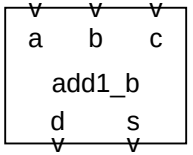


Figure 2: A 1-bit adder as a black-box (entity)

The VHDL code for such an entity is typically stored in a file **add_1b.vhd** and can have the following form:

```
-- this is a comment
-- VHDL is NOT case sensitive
-- To emphasize, the key words are capitalized or in bold
-- this is a black-box describing 1-bit adder
ENTITY add_1b IS
  PORT (a, b, c : IN   bit ;
        s, d    : OUT  bit ) ;
END add_1b ;
```

Memorize the syntax and read comments in the program. With signals we have associated data types. The simplest **data type** of a digital signal is called **bit**. As expected, a signal of the type **bit** takes on just two values, 0 and 1.

Often, it is convenient to be able to assign more than two values to a digital signal, for example, we might need the high-impedance value, 'Z', or a "don't care" condition. In such cases we replace the **bit** type with a **std_logic** data type which is available in a **library IEEE**. In most simple situations, however, we can use the **bit** type in place of the **std_logic** type.

Example: an entity for a 2-to-4 decoder

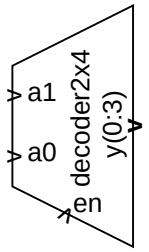


Figure 3: A 2-to-4 decoder as an entity.

```
LIBRARY IEEE
USE ieee.std_logic_1164.all

ENTITY  decoder2x4 IS
   PORT (a0, a1, en : IN  std_logic ;
                 y  : OUT std_logic_vector(0 to 3) ) ;
END decoder2x4 ;
```

Note the following elements introduced in the above program:

- We refer to all components of the library `ieee.std_logic_1164` which specifies the std_logic data type and operations performed on signals of this type.
- The entity called decoder2x4 has three input ports, a1, a0 and en, and four output ports, y(0), y(1), y(2), y(3) arranged in a single bus, that is, in a vector of signals, y(0:3).

- The **std_logic_vector** (and also **bit_vector**), is a predefined array type of **std_logic** (or **bit**), the **range** "0 **to** 3" specifies the array size.
- The range of vectors must be constant, that is the arrays are static.
- The **range** expression, e.g, "0 **to** 3" implicitly uses the signals of the type **integer**.

Architecture

- An architecture defines a body for the component entity and describes either its structure or behaviour.
- It is possible to associate a number of architectures with a given entity.

Consider two possible architectures for a 1-bit adder entity as in Figure 4:

```
-- dataflow architecture for  add_1b
ARCHITECTURE  d_flow_a  OF  add_1b  IS
BEGIN
  s <= a XOR b XOR c ;  -- a SIGNAL assignment
  d <= (a AND b) OR ((a OR b) AND c) ;
END  d_flow_a ;
```

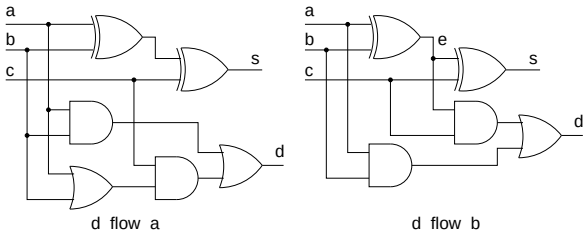


Figure 4: Two possible implementations of a 1-bit adder

This is our first VHDL architecture therefore note the following:

- the architecture name which refers to the entity name,
- a signal assignment operator '<=' ,
- logic operators, XOR, OR and AND,
- parentheses, which specify the order of operation.

Another possible architecture for the `add_1b` entity:

```
-- another dataflow architecture for  add_1b
 ARCHITECTURE  d_flow_b  OF  add_1b  IS
   SIGNAL e : std_logic ; -- internal signal declaration
 BEGIN
   e <= a XOR b ;
   s <= e XOR c ;
   d <= (a AND b) OR (e AND c) ;
 END  d_flow_b ;
```

A new element introduced in the `d_flow_b` architecture is an internal **signal** `e` which is specified as being of the type std_logic. The internal signals are always bi-directional and are used in the description of the circuit.
The complete VHDL program for the above 1-bit adder specification can look as follows

```
 LIBRARY ieee ;
 USE ieee.std_logic_1164.all;

 ENTITY tst1 IS
   PORT (a, b, c : IN   std_logic ;
         s, d    : OUT  std_logic ) ;
 END tst1 ;

 ARCHITECTURE  a1  OF  tst1  IS
 BEGIN
   s <= a XOR b XOR c ;  -- a SIGNAL assignment
   d <= (a AND b) OR ((a OR b) AND c) ;
 END  a1 ;

 ARCHITECTURE  a2  OF  tst1  IS
  SIGNAL e : std_logic ; -- internal signal declaration
 BEGIN
   e <= a XOR b ;
   s <= e XOR c ;
   d <= (a AND b) OR (e AND c) ;
 END  a2 ;
```

2.2 Circuit Specification Styles

In general, using VHDL, the internal details of a digital device are specified by an architecture body using the following three component specification styles, or their combination:

As a set of **interconnected components** to represent the **structure** of the top-level component.

As a set of **concurrent** assignment statements to represent the **data flow** inside the component.

As a set of **sequential** assignment statements to represent the **behaviour** of the component.

In a general descriptive sense we will be using terms like, digital device, circuit, block or component interchangeably. In VHDL all these terms are replaced by “component”.

2.2.1 Structural Style of Component Specification

In this style of specification a component is described as a set of other interconnected components. Each constituent component is a black box with an unspecified, at this stage, function or behaviour, but with precisely defined ports.

In the declarative part of the architecture we specify input-output ports of all components used in the architecture body in a way identical to the respective entity declarations for these components. The components may already exist in libraries, or can be specified later.

Consider a structural architecture of a 1-bit adder shown in Figure 5

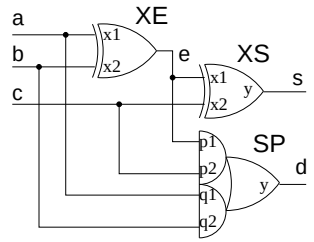


Figure 5: Structural representation of a 1-bit adder

```
-- structural architecture for  add_1b
ARCHITECTURE  struct1  OF  add_1b  IS
  COMPONENT  xor2  -- 2-input exclusive-or
    PORT( x1, x2 : IN  std_logic ;
                y : OUT std_logic ) ;
  END COMPONENT ;
  COMPONENT  aor22  -- sum of two 2-input products
    PORT( p1, p2, q1, q2 : IN  std_logic ;
                       y : OUT std_logic ) ;
  END COMPONENT ;
  SIGNAL  e : std_logic ;
BEGIN
  XE: xor2  PORT MAP (a, b, e) ;
  XS: xor2  PORT MAP (e, c, s) ;
  SP: aor22 PORT MAP (c, e, b, a, d) ;
END struct1 ;
```

Note that

- In the architecture body the library components are **instantiated** as many times as specified by the schematic describing the architecture using a **port map** component instantiation statement.

- Each **component instantiation statement** is labelled as its schematic equivalent. In the example, two different library components, “xor2” and “aor22” are instantiated three times in total, as components “XE”, “XS” and “SP”.
- Interconnections between components are specified by the **port map** statement. For it to work, every net in the schematic, that is, all external and internal signals, must be assigned a name.
- These names are specified either in the entity declaration as ports, or in the architecture as signals, e.g., the signal `e`.
- Every **port map** statement is associated by positions with the respective component declaration.
- For example, the net `e` connects the output (position 3) of the component “XE” with an input (position 1) of the component “XS” and an input (position 2) of the component “SP”.

As expected, the structural component specification describes precisely the internal structure of the component, but says noting about it behaviour. It can be, of course, inferred if the behaviour of the constituent components is known.

2.2.2 Dataflow Style of Component Specification

In this style of specification, the flow of data through a digital device is expressed using a variety of the **concurrent signal assignment statements**.

The architecture of the component does not contain an explicit description of its internal structure.

The dataflow style is perhaps the easiest to obtain the correct synthesizable code. It is similar to the concept of the “register transfer” description.

Consider as an example a simple combinational circuit presented in Figure 6 which will be described using the dataflow style.

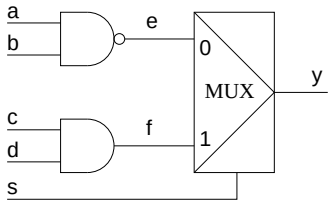


Figure 6: A simple combinational circuit

The circuit consists of three simple combinational circuits, namely, a NAND gate, an AND gate and a 2-to-1 multiplexer. In the VHDL code a concurrent assignment statement will be associated with every gate and the multiplexer.

```
ARCHITECTURE dataflow OF combD IS
  SIGNAL e, f : std_logic ;
BEGIN
  y <= e WHEN s = '0' ELSE f ; -- statement 1
  e <= a NOR b ;               -- statement 2
  f <= c AND d ;               -- statement 3
```

```
END combD ;
```

There are three concurrent statements in the program which describes the flow of data through the circuit. The order of these statements is unimportant, because these statements are executed/interpreted **concurrently**. Details of the concurrent statements will be discussed in the subsequent sections.

2.2.3 Behavioural Style of Component Specification

The behavioural style of specification is expressed by **sequential statements** executed/interpreted in the **specified order**. The sequential statements are enclosed inside a **process** statement which by itself is a concurrent statement.
Consider as an example an architecture of a two-to-four decoder as in Figure 7

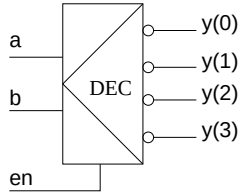


Figure 7: A 2-to-4 decoder

```
ARCHITECTURE behav OF dec2to4 IS
BEGIN
  PROCESS (a, b, en)
    VARIABLE aB , bB : std_logic ;
  BEGIN
    aB := NOT a ;                 -- statement 1
    bB := NOT b ;                 -- statement 2
    IF (en = '1') THEN            -- statement 3
      y(0) <= NOT ( aB AND bB ) ; -- statement 4
      y(3) <= NOT ( a  AND b ) ;  -- statement 5
      y(1) <= NOT ( aB AND b ) ;  -- statement 6
      y(2) <= NOT ( a  AND bB ) ; -- statement 7
    ELSE
      y <= "1111";                -- statement 8
    END IF ;
  END PROCESS ;
END behav ;
```

Note that

- In the above example we have introduced the **process** statement, which is a concurrent statement in itself, but is used to encapsulate sequential statements, like the **if** statement. We will discuss other sequential statements in the subsequent sections.
- The list of signals specified with the **process** statement constitutes a **sensitivity list**. During execution of the VHDL program, the process is invoked whenever an event occurs on any signal from the sensitivity list. Typically, all input signals should be on the sensitivity list.

In this example we have also introduced **variables**. Variables are similar to **signals** , however:

- **Variables** can be declared only within a process and they are local to that process. Signals cannot be declared inside a process.
- The assignment operator for variables is the := symbol.
- In the context of component modelling, the variables are assigned values instantaneously, as opposed to signals which are assigned values after a delay (default, or specified explicitly).
- In the context of synthesis, signals represent electrical nets, variables are not directly represented in the circuit.

2.2.4 Mixed Style of Component Specification

It is obvious that all specification styles can be mixed. Some problems are easier to describe in a particular way, namely, structural, dataflow or behavioural. We will study this issue in some depth in the subsequent sections.
As an example of a mixed specification style let us consider a combinational circuit as in Figure 8

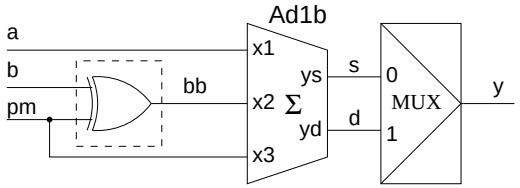


Figure 8: A simple combinational circuit modelled using a mixed style.

There are three components of the circuit: an XOR gate, a 1-bit adder, and a 2-to-1 multiplexer. In order to illustrate the concept of the mixed style of component specification, the VHDL description of the circuit will consist of three different-style concurrent statements:

- a structural port map statement for the 1-bit adder,
- a process statement to describe the behaviour of the XOR gate,
- a concurrent assignment statement to describe the data flow through the multiplexer

```
LIBRARY IEEE
USE ieee.std_logic_1164.all

ENTITY  CombCircMx IS
  PORT (a, b, pm, sd : IN  std_logic ;
                  y   : OUT std_logic ) ;
END  CombCircMx ;

-- mixed mode  architecture for CombCircMx
ARCHITECTURE  MxdMd  OF  CombCircMx  IS

  COMPONENT add1bit
    PORT( x1, x2, x3 : IN  std_logic ;
              ys, yd : OUT std_logic ) ;
  END COMPONENT ;

  SIGNAL bb, s, d : std_logic ;
```

```
BEGIN
  -- Structural statement
  AD1b: add1bit PORT MAP (a, bb, pm, s, d) ;

  -- dataflow statement
  y <= s WHEN  sd = '0' ELSE d ;

  -- behavioural statement
  PROCESS (b, pm)
  BEGIN
    IF (pm = '0') THEN
      bb <= b ;
    ELSE
      bb <= NOT b ;
    END IF ;
  END PROCESS ;
END  MxdMd ;
```


2.2.5 A clock generator

In this section we will discuss and example of a VHDL program intended not for synthesis, but for generation of waveforms, a single clk of a period 100 ns. Such a generator can be connected to another VHDL component using a component instantiation statement. This open a very important area of modelling and testing using VHDL specifications.

```
 --- a clock generator
LIBRARY ieee ;
USE ieee.std_logic_1164.all;

ENTITY clk_gen IS
  PORT ( clk : OUT  std_logic ) ;
END clk_gen ;

ARCHITECTURE  gen  OF  gen3b  IS
  SIGNAL   ck :  std_logic := '0';
  CONSTANT halfperiod : TIME := 50 ns ;
BEGIN
  PROCESS (ck)
  BEGIN
    ck <=  NOT ck AFTER  halfperiod ;
  END PROCESS ;
  clk <= ck ;
END  gen ;
```

Note that

- The internal signal ck is initialised to a value '0' .
- We use a constant of the type **time** equal to 50 ns
- The generator generates a waveform of the 100 ns period.

2.3 Summary

The following introductory VHDL concepts have been discussed in this section:

- Component
  - **entity** — component I/O ports (interface),
  - **architecture** — component specification.
- Basic data types
  - **bit** and **std_logic** — single signals,
  - **bit_vector** and **std_logic_vector** — array os signals, signal busses.
  - The **range** expression.
- Three basic types of VHDL statements and related component specification styles
  - concurrent statements — dataflow specification
  - sequential statements — behavioural specification
  - interconnection of components (**port map** — component instantiation statement) — structural specification.
- **Signals** — external and internal. **Variables**.
- A waveform generator.

4 Basic Features of VHDL for Synthesis

4.1 Libraries and Packages

A **package** consists of the *package declaration* and a *package body* and contains definitions of objects, procedures, functions, etc. that can be used in a VHDL description. An entity or architecture may **not** be defined in a package, therefore, a package does not represent a circuit. Packages are typically grouped in the libraries, and the general form to include a package in a VHDL description is of the following form:

```
library  libr ;
use  libr.package.selection ;
```

4.1.1 Fundamental packages

Packages STANDARD and TEXTIO are predefined and need not to be declared. They typically resides in the library **std**.
Package STANDARD predefines a number of types, subtypes, and functions. Some of the declarations from this package are as follows:

```
package standard is
   type boolean is (false,true);
   type bit is ('0', '1');
   type character is ( ...  ) ;
         ...
   type integer is range -2147483648 to 2147483647;
   type   real  is range -1.0E308 to 1.0E308;
   type   time  is range -2147483647 to 2147483647
        ...
```

Operations and functions defined for the above types are specified in the body of the package standard. Such a body is usually invisible.

Package TEXTIO contains declarations of types and subprograms that support formatted I/O operations on text files.

The **library ieee** contains a number of packages related to the std_logic types and we will often refer to packages from the library.

The **library work** refers to the directory containing the design under specification and need not to be declared.

4.2 Types

Every object in a VHDL source, that is, constants, signals and variables, needs to be declared and needs to be of a specific type. A type is characterised by a set of values and a set of operations. For the purpose of synthesis we can concentrate on the following two classes of types:

4.2.1 Scalar types

Scalar types consist of *enumeration* types, *integer* types, *physical* types, and *floating* point types.

Enumeration types Basic enumeration types are specified in the package **standard**. We have been already using the 9-value std_logic type specified in the package **std_logic_1164** in the following way:

```
PACKAGE std_logic_1164 IS
   --        logic state system  (unresolved)
   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
                      );
```



```
SUBTYPE std_logic IS resolved std_ulogic;
```

You can see that std_logic is a subtype of the std_ulogic. The resolution function is called when we have two concurrent assignment statements to any signal of type std_logic in order to determine the final value of the signal.

A relevant constant can be defined, if needed. in the following way:

```
constant  HiImp : std_logic  := 'Z' ;
```

A user can easily define any enumeration type, for example:

```
type  states  is  (init, st1, st2, final);
```

A signal can now be defined as:

```
signal  stt : states ;
```

Integer types are predefined in the package standard covers the range of values represented by a 32-bit twos complement number. It is possible to define integer types with different ranges, for example

```
type  integer0_15  is  range 0 to 15 ;
```

Any object of type `integer0_15` can only contain integer values in the range specified.

Floating-point types are predefined in the package standard and it is possible, as in the integer type case, to define floating-point types located within a specified range. This type is not normally used in synthesis.

Physical types represent relations between quantities. The type **time** is predefined in the following way:

```
type time is range -2147483647 to 2147483647
   units
     fs;
     ps = 1000 fs;
     ns = 1000 ps;
     us = 1000 ns;
     ms = 1000 us;
     sec = 1000 ms;
     min = 60 sec;
     hr = 60 min;
   end units;
```

The type time is usually used to model the delay of circuits, and in generation of waveform. It is not normally used in synthesis.

4.2.2 Composite types

Array types An array is a composite object consisting of elements of the same subtype. The array type can be constrained or unconstrained.

For the **constrained** array type, the number of elements and the name of the elements (the index) are defined and fixed in the type definition. For example:

```
type  byte  is array (7 downto 0) of std_logic};
constant  ten : byte := "00001010" ;
signal     a8 : byte ;
```

Individual elements of the array object can now be referred to using indexing, for example, `ten(3)`.

It is possible to refer to slices of the array, for example `ten(4 downto 1)` which value is ”0101”.

In the **unconstrained** array type, the number of elements and the name of the elements in not defined in the type definition. Only the subtype of index is specified. For example, the pre-defined **bit_vector** type:

```
type bit_vector is array (natural range <>) of bit;
```

The index range is usually constrained in the object declaration

In order to define a valid object of an unconstrained array type, we need to constrain the index range. This is normally done on the object declaration, for example:

```
constant nine: bit_vector(5 downto 0) := "001001";
```

Using the unconstrained array type we can define arrays of vectors as in the following example, which specifies the truth table of a 1-bit adder

```
TYPE arr_vec IS ARRAY (natural range <>)
            OF std_logic_vector(1 downto 0);
   CONSTANT add1bit : arr_vec(0 to 7) := (
   -- d s      abc
   ----------------- the truth table of a 1-bit adder
     "00",  -- 0  0
     "01",  -- 1  1
     "01",  -- 2  1
     "10",  -- 3  2
     "01",  -- 4  1
     "10",  -- 5  2
     "10",  -- 6  2
     "11"); -- 7  3
```

We can also specify multi-dimensional array types, for example:

```
type matrix is array (natural range <>, -- rows
                      natural range <>) of bit ;
```

The first index specifies the row number, the second – column number.

Using a multi-dimensional array the previous definition of the truth table of the 1-bit adder can be transposed into the following more compact form.

```
TYPE arr2d IS ARRAY (natural range <>,
                     natural range <>) OF std_logic;
CONSTANT faddTT : arr2d(1 downto 0, 0 to 7) := (
 --0123 4567
  "0001 0111" ,  --1  d
  "0110 1001" ); --0  s
```

In the library **ieee.std_logic_arith** there are two types defined which are similar to `std_logic_vector` , namely

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

Arithmetic operators are defined for these two types, which can be easily converted to the `std_logic_vector` type.

Record types A record type defines a collection of values which can be of different types. Consider for example the following definition:

```
type date is
  record
    day : integer range 1 to 31 ;
    month : month_name ;
    year : integer range 0 to 4000 ;
end record ;
```

The elements `day` and `year` are integers with the appropriately restricted ranges.

The element `month` is of type `month_name` and it is assumed that such a enumerates the names of all months.

The elements of a record type can be of any type, but cannot be an unconstrained array.

Consider as an example the following object of type `date` specified above:

```
constant DDay :  date := (6, June, 1944);
```

Individual elements of a record object can be accessed with a selected name. A selected name consists of the object name, followed by a dot (.) and the element name, for example:

`DDay.year` selects the year field out of `DDay` and returns the integer value 1944.

4.2.3 Type conversion

A VHDL expression of a specific type can be converted into a closely related type using the following conversion mechanism:

```
<target_type> ( < expression > )
```

For example, if we have an expression of the type unsigned, we can use the following conversion function:

```
std_logic_vector (<unsigned_expression>)
```

because the two types are closely related.
What will not work, for example, are the following conversions

```
bit_vector (<std_logic_vector_expression>)  -- error
integer (<unsigned_expression>)  -- error
```

because such types are not closely related. In such cases, the libraries provide specialised conversion functions and we can write:

```
to_bitvector (<std_logic_vector_expression>)
conv_integer (<unsigned_expression>)  -- error
```

For details you have to consult the libraries.

4.3 Objects

An object in VHDL is represented by a name and contains a value of a specific type. For our purpose we can group objects in three informal groups:

Constants, Signals, Variables .

The declaration of a constant, signal or variable assigns a name and a type to the object.

In addition a declaration of a constant assigns a value to the constant which cannot be changed.

Optionally a declaration of a signal, or variable can also assign an initial value to the object. This value is usually ignored during synthesis, but can be used in the waveform specification.

Signals represent wires (or nets) and their value are changed using the signal assignment statements (<=). Assignments to signals are not immediate, but are scheduled to be executed after a delta delay.

Variables can be declared and used only in processes, functions and procedures. Assignments to variable (using '`:=`' operator), as opposed to signal assignments, are immediate.

A loop variable does not have to be declared and it gets its type and value from the specified range in the iteration scheme. For example:

```
for  i  in  7 downto 0  loop
   c(i+1) <=  a(i) AND b(i) ;
end loop ;
```

In the example, the loop variable `i` is a integer with values 7, 6, ... , 0.

A loop variable can only be used inside the loop, and there can be no assignments to the loop variable.

For synthesis, the range specified for the loop variable must be a compile-time constant, otherwise the construct is not synthesizable.

Ports and Generics are declared inside the **entity** specification.

A **port** is an interface terminal of an entity and is an object similar to a signal, that is, its declaration assigns the name, type, and optionally an initial value.

In addition, a port has a **direction** property which indicates the possible information flow through the port. Possible directions are **in**, **out**, **inout** and **buffer**, where **inout** and **buffer** describe a bidirectional port.

A port can be used in the architecture as any other signal with restrictions flowing from its directionality: an input port cannot be assigned to, and the output port cannot be used in an expression.

Generics are parameters specified inside the component entity declaration. They are typically used to pass information about the sizes of signal ports and to specify component's timing parameters. Generics are declared in a generic list.

Consider the following typical example

```
ENTITY  combNxM  IS
  GENERIC ( N : integer := 8 ; M : integer := 5  ) ;
     PORT ( X :  in bit_vector (N-1 downto 0) ;
            Y : out bit_vector (1 to M)) ;
end combNxM ;
```

The generics `N` and `M` can now be used inside the **entity**, to define the size of ports, and in the related the **architecture**.

An important feature of generic specification is that the values assigned in the entity declaration can be **overwritten by a generic map statement** in the component instantiation of the entity.

```
C1 : combNxM  GENERIC MAP (N => 16, M => 10)
                 PORT MAP (X => C1X, Y => C1Y) ;
```

A **generic map** statement is used to specify the values of parameters of the instantiated components, in a way similar as the a **port map** statement specifies the name of port signals.

4.4 Basic operations and operators

As it has been already said, a **type** is characterized by a set of values and a set of operations. The set of operations of a type the basic operations and the predefined operators.
A **basic operation** is an operation that is inherent in, among others, a numeric literal, a string literal, a bit string literal, an aggregate, or a predefined attribute.

4.4.1 Literals

Literals are use to define types and as constants in expressions. Examples include:

- Character Literals: `'a' '0' 'Z' '%'#`
  Character literals contain only a single character, and are single quoted.
- String Literals:
  `"ZX--1"  "10101" "hallo!"  "!@#$%^&*"` String literals are double quoted and contain an array of characters.
- Bit String Literals: `B"1011_0101"  X"B5" O"2_65"`
  Bit string literals are double quoted and contain an array of the characters 0 and 1, and are preceded by a base specifier, `B` (binary), `O` (octal), or `X` (hexadecimal). Bit string literals can contain underscores which are used only for readability.
- Decimal Literals represent **integer** or **real** values:
  `-525.3  -5.253E2  32  32_000 0.32E-5`
- Based Literals are also integer or real values, but are written in a based form `<base>#<digits>#[E<digits>]`, e.g.:
  `2#10101#  4#32_13#  12#B2A#  3#201#E+2` (= 171)

Note the difference between the bit literal, say, `O"47"` which is an array of bits, and corresponding octal literal `8#47#` which is an integer (or real) equal to decimal 39.

- Physical Literals: `14 ns  3.3 V  47 pF`
- Identifiers

4.4.2 Aggregates

An aggregate is a basic operation combining values into a composite value of a record or array type:

```
( expression , expression , ... )
```

We have been using aggregates already, mainly in definitions of constants, see sec. 4.2.2, but they can be conveniently used in the signal assignment statements. The length of the aggregate must match the length of the target object, for example

```
SIGNAL  a, b : bit_vector (1 to 2) ;
SIGNAL  c, d : bit_vector (1 to 3) ;
 ...
  c <= (a, b(1) AND b(2)); -- error, length 2, 3 is expected
  d <= (a(1), a(2), b(1) AND b(2)) ; -- correct
```


Finally, we consider a description of a multi-bit multiplexer using an aggregate which arguable is the most elegant:

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all;
ENTITY  mux3 IS
  GENERIC ( N : integer := 4 ; M : integer := 3 ) ;
  PORT (a, b, c : IN  std_logic_vector (N-1 downto 0) ;
              s : IN  integer range 1 to M ;
              y : OUT std_logic_vector (N-1 downto 0) ) ;
END   mux3 ;
ARCHITECTURE  aggr  OF   mux3  IS
  TYPE arr2d IS ARRAY (natural range <>)
                OF std_logic_vector(a'RANGE);
  SIGNAL yy : arr2d(1 to M) ;
BEGIN
  yy <= (a, b, c) ; -- an aggregate
   y <= yy(s) ;
END  aggr ;
```

Please refer to the language manual for a full description of aggregates.

4.4.3 Operators

Operators are **predefined** for objects of the predefined types, like **bit**, **integer**, etc., and **overloaded** in the libraries for objects of the new types, like **std_logic**, and **unsigned**. Operators are grouped in classes which are listed in order of increasing precedence:

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

The shift operators are predefined for the **bit_vector** type and are not overloaded neither in the **ieee.std_logic_1164** nor in **ieee.std_logic_arith** libraries for the type **std_logic_vector**. However, in the ieee.std_logic_arith library there are two shift functions

`shl(arg, count) , shr(arg , count)`

where `arg` can be `unsigned` or `signed`, and `count` must be `unsigned`.

A concatenation operator `&` combines a single elements or array slices into an array slice. In many cases it works in a way similar to aggregates. Consider the following example:

```
SIGNAL  a, b : bit_vector (1 to 2) ;
SIGNAL  c, d : bit_vector (1 to 8) ;
 ...
  c(2 to 7) <= a & b(1) AND b(2) & B"10";
```

Note that it is more flexible than a corresponding expression with aggregates.

One way of warning regarding complex arithmetic operations. If we have, for instance, in our VHDL code a statement like

```
a <= b * c
```

then the synthesizer will synthesize such a statement using unstructured, that is, “random” logic, which typically results in a big circuit consisting of lots of gates. If we have a particular way of implementing such an operation, say, a bit-serial implementation, in mind, we have to write a suitable VHDL code which results in such an implementation.

4.5 Attributes

Attributes play an important role in VHDL coding facilitating the flexible style of writing VHDL specification.
Attributes denote values, functions, types, and ranges associated with various kinds of named entities. We will consider first attributes which describe values and ranges of objects of a specific type. If `t` describe type, then the related attributes result in the following values of type `t`:

`t'` **left**	—	the left bound of `t`
`t'` **right**	—	the right bound of `t`
`t'` **high**	—	the upper bound of `t`
`t'` **low**	—	the lower bound of `t`
`t'` **range**	—	the range of `t`
`t'` **reverse_range**	—	the reverse range of `t`
`t'` **length**	—	number of values in the range `t`

For example:

```
signal vx : bit_vector (8 downto 3) ;
 ...
  vx'LEFT    -- returns integer  8
  vx'RIGHT   -- returns integer  3
  vx'LENGTH  -- returns integer  6
  vx'RANGE   -- returns  range   8 downto 3
```


References

[1] IEEE Std 1076-2002, *1076 IEEE Standard VHDL Language Reference Manual*, May 2002.

[2] K.-C. Chang, *Digital Design and Modelling with VHDL and Synthesis*. IEEE Computer Society Press, 1997.

[3] IEEE Std 1076.6-1999, *IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis*, September 1999.

[4] IEEE Std 1076.3-1997, *IEEE Standard VHDL Synthesis Packages*, March 1997.

[5] Mentor Graphics Corporation, *LeonardoSpectrum HDL Synthesis*, 2001. in: /sw/leonardo_spectrum/doc/hdl_syn.pdf.