Fundamental VHDL units

As depicted in Figure 2.1, a standalone piece of VHDL code is composed of at least three fundamental sections:

- LIBRARY declarations: Contains a list of all libraries to be used in the design. For example: ieee, std, work, etc.
- ENTITY: Specifies the I/O pins of the circuit.
- ARCHITECTURE: Contains the VHDL code proper, which describes how the circuit should behave (function).

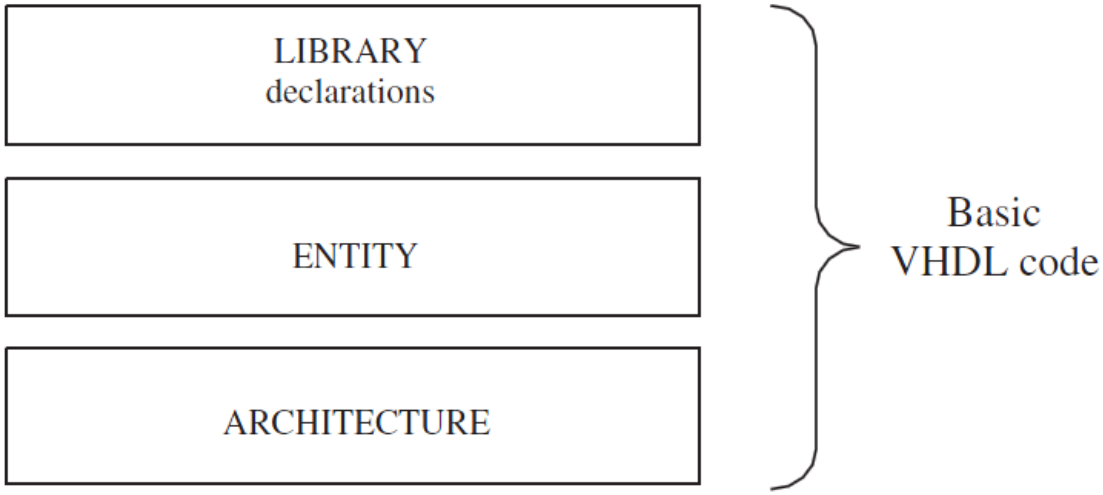

Figure 2.1 Fundamental sections of a basic VHDL code.

Library Declarations

To declare a LIBRARY (that is, to make it visible to the design) two lines of code are needed, one containing the name of the library, and the other a use clause, as shown in the syntax below.

```
LIBRARY library_name;
USE library_name.package_name.package_parts;
```

At least three packages, from three different libraries, are usually needed in a design:

- ieee.std_logic_1164 (from the ieee library),
- standard (from the std library), and
- work (work library).

Their declarations are as follows:

```
LIBRARY ieee;                          -- A semi-colon (;) indicates
USE ieee.std_logic_1164.all;           -- the end of a statement or
```

```
LIBRARY std;                          -- declaration, while a double
USE std.standard.all;                 -- dash (--) indicates a comment.

LIBRARY work;
USE work.all;
```

The purpose of the three packages/libraries mentioned above is the following: the std_logic_1164 package of the ieee library specifies a multi-level logic system; std is a resource library (data types, text i/o, etc.) for the VHDL design environment; Package standard of library std defines BIT, BOOLEAN, INTEGER, and REAL data types; and the work library is where we save our design (the .vhd file, plus all files created by the compiler, simulator, etc.).
Indeed, the ieee library contains several packages, including the following:

- Package std_logic_1164 of library ieee: Defines STD_LOGIC and STD_ULOGIC data types.
- Package std_logic_arith of library ieee: Defines SIGNED and UNSIGNED data types, plus several data conversion functions, like conv_integer(p), conv_unsigned(p, b), conv_signed(p, b), and conv_std_logic_vector(p, b).
- Packages std_logic_signed and std_logic_unsigned of library ieee: Contain functions that allow operations with STD_LOGIC_VECTOR data to be performed as if the data were of type SIGNED or UNSIGNED, respectively.

Entity

An **ENTITY** is a list with specifications of all input and output pins (**PORTS**) of the circuit. Its syntax is shown below.

```
ENTITY entity_name IS
   PORT (
      port_name : signal_mode signal_type;
      port_name : signal_mode signal_type;
      ...);
END entity_name;
```

An entity always starts with the keyword **ENTITY**, followed by its name and the keyword **is**. Next are the port declarations using the keyword **PORT**. An entity declaration always ends with the keyword **END**, optionally [] followed by the name of the entity.

- The **entity_name** is a user-selected identifier
- **port_name** consists of a comma separated list of one or more user-selected identifiers that specify external interface signals.
- **Signal_mode**: is one of the reserved words to indicate the signal direction:
  - **in** – indicates that the signal is an input
  - **out** – indicates that the signal is an output of the entity whose value can only be read by other entities that use it.
  - **buffer** – indicates that the signal is an output of the entity whose value can be read inside the entity's architecture
  - **inout** – the signal can be an input or an output.

- **Signal_type:** a built-in or user-defined signal type. Examples of types are bit, bit_vector, Boolean, character, std_logic, etc. Data types will be discussed in detail in next section.

Example: Let us consider the NAND gate of Figure 2.. Its ENTITY can be specified as:

```
entity nand_gate is
port (a, b: in bit;
      x: out bit);
end entity nand_gate;
```

Figure 2.2 NAND gate.

The meaning of the ENTITY above is the following: the circuit has three I/O pins, being two inputs (a and b, mode IN) and one output (x, mode OUT). All three signals are of type BIT. The name chosen for the entity was nand_gate.

Architecture

The architecture body specifies how the circuit operates and how it is implemented. Its syntax is the following:

```
ARCHITECTURE architecture_name OF entity_name IS
    [declarations]
BEGIN
    (code)
END architecture_name;
```

As shown above, architecture has two parts: a declarative part (optional), where signals and constants (among others) are declared, and the code part (from BEGIN down). Like in the case of an entity, the name of an architecture can be basically any name (except VHDL reserved words), including the same name as the entity's.

Example: Let us consider the NAND gate of Figure 2.2 once again.

```
architecture myarch of nand_gate is
begin
x <= a nand b;
end architecture myarch;
```

The meaning of the ARCHITECTURE above is the following: the circuit must perform the NAND operation between the two input signals (a, b) and assign (''<='')

the result to the output pin (x). The name chosen for this architecture was myarch. In this example, there is no declarative part, and the code contains just a single assignment.

An entity or circuit can be specified in a variety of ways, such as **behavioral**, **structural** (interconnected components), or a **combination of the above**. The **behavioral** level that describes a system in terms of what it does (or how it behaves) rather than in terms of its components and interconnection between them. A behavioral description specifies the relationship between the input and output signals. The **structural** level, on the other hand, describes a system as a collection of gates and components that are interconnected to perform a desired function.