Chapter Three: VHDL Fundamentals

I must create a system, or be enslav’d by another man’s; I will not reason and compare: my business is to create.

“William Blake”

2.1 VHDL Design Units

One unique property of VHDL compared to other hardware languages is the concept of the Design units. Design units are independent segments of VHDL code that can be compiled separately and stored in library. There are five types of design units in VHDL: entity, architecture, configuration, package and package body. Entity and architecture are mandatory for a design but the others are optional.

2.1.1 Entity

We elaborated top-down design methodology in chapter one and stated that the main design idea has to be broken down into smaller modules. These abstract blocks are called components that can be designed individually and they can be put together using bottom-up method and constitute the main design. In this case each one of the blocks can be treated as an entity. An entity is a black box with declaration of inputs and outputs. An example of an entity is given below:

```
entity full_adder is
  port (  
a : in   bit;
b : in   bit;
cin : in  bit;
sum : out bit;
carry : out bit
  );
end full_adder;
```

which defines a full adder with three input ports and two output ports. Each design must have at least one entity and one corresponding architecture that specifies the external specification of the design. Each entity has a names assigned to it along with a port list. Each port has a name, direction and type.
2.1.2 Architecture

An architecture statement defines the structure or description of a design and is bounded with an entity. The syntax for VHDL architecture is as follows. As you see the architecture is bounded with the entity that was defined before. VHDL allows an entity to have multiple architectures. Since architecture describes what is inside an entity, it can be written in different ways by different designers. Some prefer dataflow while the others may prefer the structural method for a design. Architecture has two parts. The first part is between the keywords architecture and begin which is the declaration part. In this section you can define the interconnection signals, other components referenced by this architecture, or constants. The second part starts after the keyword begin that includes the statements and assignments and structure of the design. The following piece of code describes the architecture of the above entity.

```vhdl
architecture full_adder_arch of full_adder is
begin
    sum   <= a xor b xor cin;
    carry <= (a and b) or (a and cin) or (b and cin);
end full_adder_arch;
```

2.1.3 Configuration

In the previous section we stated that an entity can have multiple architectures. The configuration statement specifies which entity is to be bounded to which architecture. This allows you to switch between different architectures for an entity in a design. The following is an example of a configuration statement. In this example we bound the above pair of entity-architecture.

```vhdl
configuration first_try of full_adder is
    for full_adder_arch
end for;
end first_try;
```
Configuration statement is optional for a design. In case if you have multiple architectures defined for an entity in a design, synthesizer always uses the most recent compiled architecture.

### 2.1.4 Package

It could be very tedious to repeat the declarations every time you need them. As a programmer you want to be able to use a piece of code over and over once you write it to save time and resources. Package provides this flexibility in VHDL and is used to store frequently used declarations among different design units.

Package may be used in its simplest form to declare components, constants, global variables, or global functions which can be made visible not only for the current design but also for any other design units that reference the package. A package can be accessed using the `use` statement, which will be explained later. A package can be identified using the keyword `package`. The following is an example of a package.

```vhdl
package conversion is
    function int_to_vector (max_size: integer; number: integer) return bit_vector;
end conversion;
```

A package can consist of two distinct parts: a package declaration and an optional package body. In the above example the function declaration is given. There might be some other declarations such as constants or variables in a package as well.

### 2.1.5 Package Body

A package body is always associated with package definition with the same name. If a package includes a function or a deferred constant a package body is needed to define the value of the constant or definition of the function. The following example gives the details of the function that was in the above package.

```vhdl
package body conversion is
    function int_to_vector (max_size: integer; number: integer) return bit_vector is
        variable return_vector: bit_vector (1 to max_size);
        variable x: integer;
    begin
        x := number;
        for i in max_size downto 1 loop
            if ((x mod 2) = 1) then
                return_vector(i) := '1';
            else
                return_vector(i) := '0';
            end if;
        end loop;
    return return_vector;
end conversion;
```
end if;
x := x / 2;
end loop;
return return_vector;
end int_to_vector;
end conversion;

2.2 Levels of Abstraction

A course description advertising the Robot Design to MIT community reads like this:

You are given a kit containing a microprocessor, LEGO blocks, batteries, motors, sensors, and wire. Your task: design and build a robot to play in a robot sporting event (details to be provided). Lectures, recitations, and lots of laboratory hours will help you in your task. You have one month.

There was no specific technological knowledge such as advanced electronics or mechanical design or soldering skills was required for this course except a programming background. The intent was not having the student reinvent the basics of robotics but rather it was enabling the students to come up with methodological approach and differentiate the levels of abstractions in design flow. Student is supposed to insulate himself/herself from the details of the implementation and think about the project implementation at higher level. All the basics were provided to the students in this case.

VHDL supports different styles of design description. These styles vary in a sense that how they relate to the underlying hardware. When we speak of the different styles of VHDL, we are actually talking about the levels of abstraction. Figure 2-1 shows four possible levels of
language abstractions: behavioral, RTL (or Dataflow), structural (sometimes called gate-level or logic) and layout.

As an example of these four levels of abstraction, consider a simple BCD counter. It is possible to describe the counter in a number of ways. At the lowest level of abstraction (layout level), we could use the layout of transistors and build gates and flip flops out of them and connect them so as logic function can be satisfied. At the next level (logic or structural) counter can be specified as a sequence of predefined logic gates and flip-flops to form the complete circuit. To describe this same circuit at a dataflow or RTL level of abstraction, we might describe the combinational logic portion of the counter (its decoding portion) using higher-level Boolean logic functions and then feed the output of that logic into a set of registers. At the behavioral level of abstraction, regardless of target technology we just define the behavior of the system using human language. In this case this could be as “This counter counts from 0 to 9 and rolls over after it reaches the highest count.

### 2.2.1 Behavioral Modeling

Behavioral modeling is the highest level of abstraction in VHDL. In this method, the behavior of the system should be defined with respect to time. The operation of time is the critical point in behavioral modeling and it distinguishes this type from the others. There are multiple of ways to represent the time in
VHDL that will be discussed in details later. The concept of time could be defined exactly with the actual delays between the events or in the order of the sequential operations. After you write the VHDL code in behavioral, it has to be synthesized to create the netlist. Synthesis is the process of translating an abstract concept (behavioral code in this case) into a less abstract form (a netlist in this case). A netlist is the translation of your code in terms of gates so the implementation tool can understand it. It is unlikely that the synthesis tool can understand your behavioral code and create the same behavior that you have defined. Today’s synthesis tools ignore timing details of the design and the delays will depend on the target device technology. A behavioral code is very similar to a software program like C++ in which smaller programs and instructions operate sequentially and talk to one another. In software programming the execution platform (which is the compiler) is located in CPU while in VHDL it is the simulation or synthesized hardware that can execute your code. A testbench with all timing equations is an example of behavioral code that can be written in combination with your code to verify the functionality. Testbenches can be understood by simulation tools and not the synthesis tools.

2.2.2 Dataflow Modeling

The highest level of abstraction accepted by the synthesis tools available in today’s marketplace is dataflow level. In dataflow modeling you define your circuit in terms of how data moves within the system. Registers are the main components in digital systems and the behavior of the system can be described as data flow among the registers. Normally we can be very specific in a design at where the registers are located but the combinatorial part of the design can be described at relatively high level and the synthesis tool will figure out how the details of the implementation would be. Sometimes this type of implementation of modeling digital systems is called RTL or Register Transfer Logic. We stated that the behavioral code should be verified by simulation but it is not synthesizable. After the simulation, design must be redefined until it can be understandable by the synthesis tools.

Most of the times hardware designers prefer to write the code at this level of abstraction since this is closer to the actual hardware components such as registers and gates.

2.2.3 Structural Modeling
In structural modeling the smaller components connect together to form the larger components and eventually the whole circuit. Designer might use this type of modeling in a very low level description of a circuit such as gates and flip flops or in a high level description such as block diagrams. Designer can use the latter case in order to break down a complex block into manageable parts. Either way, some internal signals are needed to provide the interconnection between the components. Using this method you can drastically simplify a design using a top-down design methodology. In structural modeling, lower level modules are components that get connected to form higher level modules. In fact structural modeling represents the design in the form of a netlist of connected components. It is very important that every module is declared before it is used. Declaration is identified using component declaration and connecting the components is done using component instantiation.

### 2.2.3.1 Component Declaration

A component declaration declares design entity interface that will be used later to form a hierarchical design. Following is an example of a component declaration.

```vhdl
component half_adder
  port (a: in bit;
b: in bit;
sum: out bit;
carry: out bit);
end component;
```

### 2.2.3.2 Component Instantiation

A component instantiation is a statement that references lower-level component in the design, in essence creating unique copy (or instance) of that component. A component instantiation statement is a concurrent statement, so there is no significance to the order in which components are referenced. The concept of concurrent and sequential statements will be covered in details in later chapters. You must, however, declare any components that you reference in either the declarative area of the architecture (before the `begin` statement) or in an external package that is visible to the architecture. Following is an example of a
component instantiation that makes an instance of the half adder that was declared in previous section. A good analogy for component instantiation would be putting an electronic IC into a socket.

```vhdl
add0: half_adder port map(a => a(0),
b => b(0),
sum => s(0),
carry => c(0));
```

### 2.3 Libraries in VHDL

A library, in general, is a collection of compiled design units that is being used to manage designs. The keyword **library** identifies a library in VHDL followed by the logical library name. A library becomes visible when a library statement is used. Following examples show how to declare libraries.

```vhdl
library IEEE;
declares IEEE library which is a library that includes conversion functions, useful types, etc.
library std_logic_1164;
declares IEEE 1164 standard library
library LSI_1100;
declares LSI ASIC library
```

### 2.3.1 Design Library

A design library is a storage area for previously compiled design units. A design library is completely implementation dependent, meaning that it depends on synthesis tool. Typically, you specify the areas to store frequently used design units such as packages and package bodies for future reference. If you don’t do so, the design units will be compiled into a default library called **work**. In case if you work with simple designs and individual source files, the library is accessible using the use statements which will be explained in the next section otherwise you should have a good understanding of how libraries are managed in the design process. Using named libraries increases productivity in design flow. The point is not to use the **work** library in different source files that are in different locations. All design units include **std** and **work** libraries by default but the contents of the work libraries might be different since they are results of
different implementations. Most of the synthesis and simulation tools follow the standard definition in VHDL that defines the **work** library to be only those design units that are included in the source file currently being compiled.

### 2.3.2 Use Statement

To use a design unit from within a library or other design units, a **use** statement must be used specifying the design unit. A **use** statement makes the referenced design unit visible to the working environment. A **use** statement starts with the keyword **use** followed by the list of the design units. The use statement is flexible and it can be used in different ways. You can address the individual components in a library or all of the items available in the library. Following examples show how to use libraries.

```vhdl
use ieee.std_logic_1164.all;
makes all components in std_logic_1164 library from ieee library visible

use work.func_package.all;
makes all components in func_package library from work library visible

use ieee.std_logic-arith.all;
makes all components in std_logic_arith library from ieee library visible

use ieee.std_logic_unsigned.all;
makes all components in std_logic_unsigned library from ieee library visible

use my_lib.reg_pack.jk;
makes jk flip flop component in reg_pack package from my_lib library visible

use my_lib.data_pack.all;
makes all components in data_pack package from my_lib library visible

use my_lib.all;
makes all components in my_lib library visible

use work.all;
makes all components in work library visible
```

The **library** statement described in the previous section is used to load a library so that its contents are available when compiling a source file. However, the **library** statement does not actually make the contents of the specified library visible to design units in the current source file. Visibility is created when you specify one or more **use** statements prior to the design units requiring access to items in the library. A good
practice is to place library statement in the beginning of the source file followed by the use statement whenever a visibility to a specific item or library or package is needed and also avoid using work library for shared packages.

2.4 Putting It Together!! VHDL Skeleton

Now that you understand the basics of VHDL, let’s put it together and write a code for a complete design of a 4 bit full adder from scratch. For the sake of example let’s build the 4 bit adder based on the full adder design in section 2.1.1. In this example the full adder design is considered as a component that is instantiated in the 4 bit adder design. Here is the complete VHDL code:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity adder_4 is
  port (a_bus : in bit_vector(3 downto 0);
         b_bus : in bit_vector(3 downto 0);
         cin: in bit;
         sum_bus : out bit_vector(3 downto 0);
         cout : out bit);
end adder_4;

architecture structural of adder_4 is

signal carry_int: bit_vector(3 downto 0);

component full_adder
  port (a : in bit;
        b : in bit;
        cin: in bit;
        sum : out bit;
        carry : out bit);
end component;

begin
  add0: full_adder port map (a => a_bus(0),
                           b => b_bus(0),
                           cin => cin,
                           a_bus(1),
                           b_bus(1),
                           cin(0),
                           sum_bus(0),
                           carry(0));
end adder_4;
```
The code starts with necessary library declarations and is followed by entity and architecture declarations. The entity part defines the design as a black box with inputs and outputs. The architecture part defines the interrelation of these inputs and outputs. In fact architecture specifies what is inside the black box. The architecture part is divided into two sections: declaration and instantiation part.

Declaration part is between the keywords architecture and begin. Here, all the signals, components, constants, variables, … must be declared. In our example, carry_int signal is defined to handle interconnection between the single bit full adders. Also the single bit full adder is defined as a component.

The instantiation part starts after the keyword begin. Four instances of the full adder are constructed and using the port map statements they are connected together. Note that full adder instantiations are concurrent statements and order does not matter. These are the statements that are executed in parallel and operate asynchronously. Concurrent statements are versus sequential statements that are executed sequentially upon the flow of the program. We will see this concept in near future.

2.5 VHDL Basic Elements
In this section we are going to discuss the basic concept in VHDL needed to start learning this language. VHDL unlike Verilog or C++ is generally a case insensitive language meaning that there is no differentiation between upper case and lower case letters. In order to have a better readability of the code, you can decide have your own rules, for instance, to write the keywords in lower case and the user defined words in upper case. All the statements in VHDL end up with semicolon. Signal assignments are done using (<=) sign. And lists of variables, signals, ports, etc. are separated with commas. There are many revisions of VHDL language but for the first time, VHDL was standardized in 1987 as IEEE 1076-1987 standard or simply VHDL-87 which is the most important version of the language. Most of the available tools in the market are still based on this version. As time went by, the first standard was analyzed by the users and various modifications were made by the experts and finally a new revision was introduced by IEEE 1076-1993 or simply VHDL-93. Unfortunately these two versions are not fully compatible and transferring a program from one to another needs slight modifications. This book does not discuss all aspects of these two versions but it mentions the differences between two standards when it is necessary.

2.5.1 Lexical Elements

A VHDL text may consist of multiple design files that are made of ASCII character set. A lexical element is a set of characters that together define a basic element of VHDL that can not be broken into smaller elements. In the following sub-sections we review all the lexical elements available in VHDL.

2.5.1.1 Comments

Comments are used to clarify the code and explain the purpose of each line or pieces of VHDL code. Comments are very useful especially when the person who reads the code is not the writer of the code. Comments are symbolized by double dash (- -) at the start of the comment line. Comments can start anywhere on a line of code and run until the end of the line.
2.5.1.2 Identifiers

Identifiers are used to name data objects so they can be referenced later by the user anywhere in the program. Simple identifiers as defined by the VHDL 87 standard may contain letters, numerals and underscores. In particular the identifier has to begin with a letter, so (4_adder) is not a valid identifier. Also graphical characters, space character, VHDL keywords, and consecutive underscores are not allowed. Note that an identifier can not end with underscore.

In the VHDL 93 standard, a new type of identifiers is defined which is called extended identifier. An extended identifier is enclosed in back slashes. Within these back slashes nearly every combination of characters, numbers, white spaces and underscores is allowed. The only thing to consider is that extended identifiers are now case sensitive. So (/test_signal/), (/TEST_SIGNAL/) are two different identifiers.

2.5.1.3 Numbers and Characters

Table 2-1 lists all acceptable characters and numbers in VHDL along with the description of each. Since all the tools and synthesizers may not support all the items in this table, refer to your VHDL reference manual.

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-9</td>
<td>digits</td>
<td>{</td>
<td>left brace</td>
</tr>
<tr>
<td>A-Z</td>
<td>uppercase Letters</td>
<td>}</td>
<td>right brace</td>
</tr>
<tr>
<td>a-z</td>
<td>lowercase Letters</td>
<td>[</td>
<td>left square bracket</td>
</tr>
<tr>
<td>+</td>
<td>plus</td>
<td>]</td>
<td>right square bracket</td>
</tr>
<tr>
<td>-</td>
<td>minus, hyphen</td>
<td></td>
<td>vertical bar</td>
</tr>
<tr>
<td>*</td>
<td>multiply, asterisk, star</td>
<td>\</td>
<td>back slash</td>
</tr>
<tr>
<td>/</td>
<td>divide, forward slash</td>
<td>:</td>
<td>colon</td>
</tr>
<tr>
<td>=</td>
<td>equal</td>
<td>&quot;</td>
<td>quotation</td>
</tr>
<tr>
<td>'</td>
<td>grave accent</td>
<td>;</td>
<td>semicolon</td>
</tr>
<tr>
<td>~</td>
<td>tilde</td>
<td>‘</td>
<td>apostrophe</td>
</tr>
<tr>
<td>@</td>
<td>exclamation</td>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>#</td>
<td>commercial at</td>
<td>&lt;=</td>
<td>less than or equal, assignment</td>
</tr>
<tr>
<td>$</td>
<td>dollar</td>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>%</td>
<td>percent</td>
<td>=&gt;</td>
<td>mapping</td>
</tr>
<tr>
<td>^</td>
<td>circumflex</td>
<td>,</td>
<td>comma</td>
</tr>
<tr>
<td>&amp;</td>
<td>ampersand</td>
<td>.</td>
<td>point, dot, period</td>
</tr>
</tbody>
</table>
2.5.1.4 Delimiters

Delimiters separate the characters or items in VHDL. As we go along the meaning of the delimiters will be described through the text. Here are a few examples to clarify:

; ; semicolon, to terminate a VHDL statement
=> => mapping sign, maps the signal to an input/output port
& & ampersand, to concatenate the bit stream

2.5.1.5 Strings

A string is a set of characters surrounded by quotations. Every string has a length that can be addressed using VHDL attributes. The concept of attributes will be explained later in details. A string with length of zero is a null string and is denoted by "". Strings can not have the length of more than one line of code but they smaller strings can be concatenated using & (ampersand) character to form larger strings.

2.5.1.6 Bit Strings

A bit string is simply a strings of digits between two quotations preceded by the radix. The radix could be one of the three letters B for binary, O for octal or X for hexadecimal.
2.5.1.7 Special Symbols