Features:

- A rich toolbox of practical FPGA design techniques at an engineer's finger tips
- Easy-to-find structure that allows the engineer to quickly locate the information to solve their FPGA design problem, and obtain the level of detail and understanding needed
- Includes a CDROM containing code, test benches and simulation files for ModelSim

This book provides a rich toolbox of design techniques and templates to solve practical, every-day problems using FPGAs. Using a modular structure, the book gives ‘easy-to-find’ design techniques and templates at all levels, together with functional code, which engineers can easily match and apply to their application.

The ‘easy-to-find’ structure begins with a design application to demonstrate the key building blocks of FPGA design and how to connect them, enabling the experienced FPGA designer to quickly select the right design for their application, while providing the less experienced a ‘road map’ to solving their specific design problem.

Written in an informal and ‘easy-to-grasp’ style, this invaluable resource goes beyond the principles of FPGAs and hardware description languages to actually demonstrate how specific designs can be synthesized, simulated and downloaded onto an FPGA. In addition, the book provides advanced techniques to create ‘real world’ designs that fit the device required and which are fast and reliable to implement. An accompanying CDROM contains code, test benches and simulation command files for ModelSim.
# Table of Contents

**SECTION 1: INTRODUCTION**  
CHAPTER 1: INTRODUCTION  
CHAPTER 2: AN FPGA PRIMER  
CHAPTER 3: A VHDL PRIMER – THE ESSENTIALS  
CHAPTER 4: DESIGN AUTOMATION AND TESTING FOR FPGAS  

**SECTION 2: APPLICATIONS**  
CHAPTER 5: INTRODUCTION  
CHAPTER 6: IMAGES AND HIGH SPEED PROCESSING  
CHAPTER 7: EMBEDDED PROCESSORS  

**SECTION 3: DESIGNER’S TOOLBOX**  
CHAPTER 8: SERIAL COMMUNICATIONS  
CHAPTER 9: DIGITAL FILTERS  
CHAPTER 10: SECURE SYSTEMS  
CHAPTER 11: MEMORY  
CHAPTER 12: PS/2 MOUSE INTERFACE  
CHAPTER 13: PS/2 KEYBOARD INTERFACE  
CHAPTER 14: A SIMPLE VGA INTERFACE  

**SECTION 4: OPTIMIZING DESIGNS**  
CHAPTER 15: ADVANCED TECHNIQUES  
CHAPTER 16: SYNTHESIS  
CHAPTER 17: BEHAVIOURAL MODELING IN VHDL  
CHAPTER 18: DESIGN OPTIMIZATION  
CHAPTER 19: VHDL-AMS  
CHAPTER 20: DESIGN OPTIMIZATION EXAMPLE: DES  

**SECTION 5: FUNDAMENTAL TECHNIQUES**  
CHAPTER 21: COUNTERS  
CHAPTER 22: LATCHES, FLIP-FLOPS AND REGISTERS  
CHAPTER 23: SERIAL TO PARALLEL & PARALLEL TO SERIAL CONVERSION  
CHAPTER 24: ALU FUNCTIONS  
CHAPTER 25: DECODERS AND MULTIPLEXERS  
CHAPTER 26: FINITE STATE MACHINES IN VHDL  
CHAPTER 27: FIXED POINT ARITHMETIC IN VHDL  
CHAPTER 28: BINARY MULTIPLICATION  
CHAPTER 29: BIBLIOGRAPHY
Introduction

This chapter of the book is not intended as a comprehensive VHDL reference book – there are many excellent texts available that fit that purpose including Mark Zwolinski’s *Digital System Design with VHDL*, Zainalabedin Navabi’s *VHDL: Analysis and modeling of digital systems* or Peter Ashenden’s *Designer’s Guide to VHDL*. This section is designed to give concise and useful summary information on important language constructs and usage in VHDL – helpful and easy to use, but not necessarily complete.

This chapter will introduce the key concepts in VHDL and the important syntax required for most VHDL designs, particularly with reference to Field Programmable Gate Arrays (FPGAs). In most cases, the decision to use VHDL over other languages such as Verilog or SystemC, will have less to do with designer choice, and more to do with software availability and company decisions. Over the last decade or so, a ‘war of words’ has raged between the VHDL and Verilog communities about which is the best language, and in most cases it is completely pointless as the issue is more about design than syntax. There are numerous differences in the detail between VHDL and Verilog, but the fundamental philosophical difference historically has been the design context of the two languages. Verilog has come from a ‘bottom-up’ tradition and has been heavily used by the IC industry for cell-based design, whereas the VHDL language has been developed much more from a ‘top-down’ perspective. Of course, these are generalizations and largely out of date in a modern context, but the result is clearly seen in the basic syntax and methods of the two languages.

Without descending into a minute dissection of the differences between Verilog and VHDL one important advantage of VHDL is
the ability to use multiple levels of model with different architectures as shown in Figure 6.

This is not unique to VHDL, and in fact Verilog does have the concept of different behavior in a single ‘module’; however, it is explicitly defined in VHDL and is extremely useful in putting together practical multi-level designs in VHDL. The division of a model into its interface part (the ‘entity’ in VHDL) and the behavior part (the ‘architecture’ in VHDL) is an incredibly practical approach for modeling multiple behavior for a single interface and makes model exchange and multiple implementations straightforward.

The remainder of this chapter will describe the key parts of VHDL, starting with the definition of a basic model structure using entities and architectures, discuss the important variable types, review the methods of encapsulating concurrent, sequential and hierarchical behavior and finally introduce the important fundamental data types required in VHDL.

**Entity: model interface**

**Entity definition**

The entity defines how a design element described in VHDL connects to other VHDL models and also defines the name of the model. The entity also allows the definition of any parameters that are to be passed into the model using hierarchy. The basic template for an entity is as follows:

```vhdl
entity <name> is
    ....
entity <name>;
```
If the entity has the name ‘test’, then the entity template could be either:

```vhdl
entity test is
end entity test;
```

or:

```vhdl
entity test is
end test;
```

## Ports
The method of connecting entities together is using PORTS. These are defined in the entity using the following method:

```vhdl
port (  ...
list of port declarations...
);
```

The port declaration defines the type of connection and direction where appropriate. For example, the port declaration for an input bit called in1 would be as follows:

```vhdl
in1 : in bit;
```

And if the model had two inputs (in1 and in2) of type bit and a single output (out1) of type bit, then the declaration of the ports would be as follows:

```vhdl
port (  in1, in2 : in bit;
        out1 : out bit
);
```

As the connection points between entities are effectively the same as those inter-process connections, they are effectively signals and can be used as such within the VHDL of the model.

## Generics
If the model has a parameter, then this is defined using generics. The general declaration of generics is shown below:

```vhdl
generic (  ...
list of generic declarations...
);
```
In the case of generics, the declaration is similar to that of a constant with the form as shown below:

\[ \text{param1 : integer} := 4; \]

Taking an example of a model that had two generics (gain (integer) and time_delay (time)), they could be defined in the entity as follows:

\[
\text{generic (}
\begin{align*}
\text{gain} : \text{integer} & := 4; \\
\text{time delay} : \text{time} & = \text{10 ns}
\end{align*}
\); \]

**Constants**

It is also possible to include model specific constants in the entity using the standard declaration of constants method previously described, for example:

\[ \text{constant : rpullup : real} := 1000.0; \]

**Entity examples**

To illustrate a complete entity, we can bring together the ports and generics examples previously and construct the complete entity for this example:

```
entity test is
  port (in1, in2 : in bit; out1 : out bit);
  generic (gain : integer := 4; time delay : time := 10 ns);
  constant : rpullup : real := 1000.0;
end entity test;
```

**Architecture: model behavior**

**Basic definition of an architecture**

While the entity describes the interface and parameter aspects of the model, the architecture defines the behavior. There are several types of VHDL architecture and VHDL allows different architectures to
be defined for the same entity. This is ideal for developing behavioral, Register Transfer Level RTL and gate Level architectures that can be incorporated into designs and tested using the same test benches.

The basic approach for declaring an architecture could be as follows:

```
architecture behaviour of test is
  ..architecture declarations
begin
  ...architecture contents
end architecture behaviour;
```

or

```
architecture behaviour of test is
  ..architecture declarations
begin
  ...architecture contents
end behaviour;
```

Architecture declaration section
After the declaration of the architecture name and before the begin statement, any local signals or variables can be declared. For example, if there were two internal signals to the architecture called sig1 and sig2, they could be declared in the declaration section of the model as follows:

```
architecture behaviour of test is
  signal sig1, sig2 : bit;
begin

Then the signals can be used in the architecture statement section.

Architecture statement section
VHDL architectures can have a variety of structures to achieve different types of functionality. Simple combinatorial expressions use signal assignments to set new signal values as shown below:

```
out1 <= in1 and in2 after 10 ns;
```

Note that for practical design, the use of the ‘after 10 ns’ is not synthesizable. In practice, the only way to ensure correct synthesizable design is to either make the design delay insensitive or synchronous. The design of combinatorial VHDL will result is
additional delays due to the technology library gate delays, potentially resulting in glitches or hazards. An example of a multiple gate combinatorial architecture using internal signal declarations is given below:

```vhdl
architecture behavioural of test is
  signal int1, int2 : bit;
begin
  int1 <= in1 and in2;
  int2 <= in3 or in4;
  out1 <= int1 xor int2;
end architecture behavioural;
```

**Process: basic functional unit in VHDL**

The process in VHDL is the mechanism by which sequential statements can be executed in the correct sequence, and with more than one process, concurrently. Each process consists of a sensitivity list, declarations and statements. The basic process syntax is given below:

```vhdl
process sensitivity_list is
  ... declaration part
begin
  ... statement part
end process;
```

The sensitivity list allows a process to be activated when a specific signal changes value, for example a typical usage would be to have a global clock and reset signal to control the activity of the process, for example:

```vhdl
process (clk, rst) is
begin
  ... process statements
end process;
```

In this example, the process would only be activated when either clk or rst changed value. Another way of encapsulating the same behavior is to use a wait statement in the process so that the process is automatically activated once, and then waits for activity on either signal before running the process again. The same process could then be written as follows:

```vhdl
process
begin
  ... process statements
  wait on clk, rst;
end process;
```
In fact, the location of the wait statement is not important, as the VHDL simulation cycle executes each process once during initialization, and so the wait statement could be at the start or the end of the process and the behavior would be the same in both cases.

In the declaration section of the process, signals and variables can be defined locally as described previously, for example a typical process may look like the following:

```vhdl
process (a) is
  signal na : bit;
begin
  na <= not a;
end process;
```

With the local signal `na` and the process activated by changes on the signal `a` that is externally declared (with respect to the process).

### Basic variable types and operators

#### Constants
When a value needs to be static throughout a simulation, the type of element to use is a constant. This is often used to initialize parameters or to set fixed register values for comparison. A constant can be declared for any defined type in VHDL with examples as follows:

```vhdl
constant a : integer := 1;
calendar b : real := 0.123;
calendar c : std_logic := '0';
```

#### Signals
Signals are the link between processes and sequential elements within the processes. They are effectively ‘wires’ in the design and connect all the design elements together. When simulating signals, the simulator will in turn look at updating the signal values and also checking the sensitivity lists in processes to see whether any changes have occurred that will mean that processes become active.

Signals can be assigned immediately or with a time delay, so that an event is scheduled for sometime in the future (after the
specified delay). It is also important to recognize that signals are not the same as a set of sequential program code (such as in C), but are effectively concurrent signals that will not be able to be considered stable until the next time the process is activated.

Examples of signal declaration and assignment are shown below:

```vhdl
signal sig1 : integer := 0;
signal sig2 : integer := 1;
sig1 <= 14;
sig1 <= sig2;
sig1 <= sig2 after 10 ns;
```

**Variables**

While signals are the external connections between processes, variables are the internal values within a process. They are only used in a sequential manner, unlike the concurrent nature of signals within and between processes. Variables are used within processes and are declared and used as follows:

```vhdl
variable var1 : integer := 0;
variable var2 : integer := 1;
var1 := var2;
```

Notice that there is no concept of a delay in the variable assignment – if you need to schedule an event, it is necessary to use a signal.

**Boolean operators**

VHDL has a set of standard Boolean operators built in, which are self-explanatory. The list of operators are and, or, nand, not, nor, xor. These operators can be applied to BIT, BOOLEAN or logic types with examples as follows:

```vhdl
out1 <= in1 and in2;
out2 <= in3 or in4;
out5 <= not in5;
```

**Arithmetic operators**

There are a set of arithmetic operators built into VHDL which again are self-explanatory and these are described and examples provided, see next page.
Comparison operators

VHDL has a set of standard comparison operators built in, which are self-explanatory. The list of operators are =, /=, <, <=, >, >=. These operators can be applied to a variety of types as follows:

```
in1 < 1
in1 /= in2
in2 >= 0.4
```

Shifting functions

VHDL has a set of six built in logical shift functions which are summarized below:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>sll</td>
<td>Shift Left Logical</td>
<td>reg &lt;= reg sll 2;</td>
</tr>
<tr>
<td>srl</td>
<td>Shift Right Logical</td>
<td>reg &lt;= reg srl 2;</td>
</tr>
<tr>
<td>sla</td>
<td>Shift Left Arithmetic</td>
<td>reg &lt;= reg sla 2;</td>
</tr>
<tr>
<td>sra</td>
<td>Shift Right Arithmetic</td>
<td>reg &lt;= reg sra 2;</td>
</tr>
<tr>
<td>rol</td>
<td>Rotate Left</td>
<td>reg &lt;= reg rol 2;</td>
</tr>
<tr>
<td>ror</td>
<td>Rotate Right</td>
<td>reg &lt;= reg ror 2;</td>
</tr>
</tbody>
</table>

Concatenation

The concatenation function XE ‘VHDL:concatenation’ in VHDL is denoted by the & symbol and is used as follows:

```
A <= '1111';
B <= '000';
out1 <= A & B & '1'; -- out1 = '11110001';
```
Decisions and loops

If-then-else

The basic syntax for a simple if statement is as follows:

```vhdl
if (condition) then
    ... statements
end if;
```

The condition is a Boolean expression, of the form `a > b` or `a = b`. Note that the comparison operator for equality is a single `=`, not to be confused with the double `==` used in some programming languages. For example, if two signals are equal, then set an output high would be written in VHDL as:

```vhdl
if (a = b) then
    out1 <= '1';
end if;
```

If the decision needs to have both the if and else options, then the statement is extended as follows:

```vhdl
if (condition) then
    ... statements
else
    ... statements
end if;
```

So in the previous example, we could add the else statements as follows:

```vhdl
if (a = b) then
    out1 <= '1';
else
    out1 <= '0';
end if;
```

And finally, multiple if conditions can be implemented using the general form:

```vhdl
if (condition1) then
    ... statements
elsif (condition2)
    ... statements
    ... more elsif conditions & statements
else
    ... statements
end if;
```
With an example:

```vhd
if (a > 10) then
    out1 <= '1';
elsif (a > 5) then
    out1 <= '0';
else
    out1 <= '1';
end if;
```

**Case**

As we have seen with the IF statement, it is relatively simple to define multiple conditions, but it becomes a little cumbersome, and so the case statement offers a simple approach to branching, without having to use Boolean conditions in every case. This is especially useful for defining state diagrams or for specific transitions between states using enumerated types. An example of a case statement is:

```vhd
case testvariable is
    when 1 =>
        out1 <= '1';
    when 2 =>
        out2 <= '1';
    when 3 =>
        out3 <= '1';
end case;
```

This can be extended to a range of values, not just a single value:

```vhd
case test is
    when 0 to 4 => out1 <= '1';
end case;
```

It is also possible to use Boolean conditions and equations. In the case of the default option (i.e. when none of the conditions have been met), then the term when others can be used:

```vhd
case test is
    when 0 => out1 <= '1';
    when others => out1 <= '0';
end case;
```

**For**

The most basic loop in VHDL is the FOR loop. This is a loop that executes a fixed number of times. The basic syntax for the FOR loop is shown below:

```vhd
for loopvar in start to finish loop
    ... loop statements
end loop;
```
It is also possible to execute a loop that counts down rather than up, and the general form of this loop is:

```vhdl
for loopvar in start downto finish loop
  ... loop statements
end loop;
```

A typical example of a for loop would be to pack an array with values bit by bit, for example:

```vhdl
signal a : std_logic_vector(7 downto 0);
for i in 0 to 7 loop
  a(i) <= '1';
end loop;
```

**While and loop**

Both the while and loop loops have an in-determinant number of loops, compared to the fixed number of loops in a FOR loop and as such are usually not able to be synthesized. For FPGA design, they are not feasible as they will usually cause an error when the VHDL model is compiled by the synthesis software.

**Exit**

The exit command allows a FOR loop to be exited completely. This can be useful when a condition is reached and the remainder of the loop is no longer required. The syntax for the exit command is shown below:

```vhdl
for i in 0 to 7 loop
  if ( i = 4 ) then
    exit;
  endif;
endloop;
```

**Next**

The next command allows a FOR loop iteration to be exited, this is slightly different to the exit command in that the current iteration is exited, but the overall loop continues onto the next iteration. This can be useful when a condition is reached and the remainder of the iteration is no longer required. An example for the next command is shown below:

```vhdl
for i in 0 to 7 loop
  if ( i = 4 ) then
    next;
  endif;
endloop;
```
Hierarchical design

Functions

Functions are a simple way of encapsulating behavior in a model that can be reused in multiple architectures. Functions can be defined locally to an architecture or more commonly in a package (discussed in the next section of this book), but in this section the basic approach of defining functions will be described. The simple form of a function is to define a header with the input and output variables as shown below:

```vhdl
function name (input declarations) return output_type is
  ... variable declarations
begin
  ... function body
end
```

For example, a simple function that takes two input numbers and multiplies them together could be defined as follows:

```vhdl
function mult (a, b : integer) return integer is
begin
  return a * b;
end;
```

Packages

Packages are a common single way of disseminating type and function information in the VHDL design community. The basic definition of a package is as follows:

```vhdl
package name is
  ...package header contents
end package;
package body name is
  ... package body contents
end package body;
```

As can be seen, the package consists of two parts, the header and the body. The header is the place where the types and functions are declared, and the package body is where the declarations themselves take place.

For example, a function could be described in the package body and the function is declared in the package header. Take a simple example of a function used to carry out a simple logic function:

```vhdl
and10 = and(a, b, c, d, e, f, g, h, i, j)
```
The VHDL function would be something like the following:

```vhdl
function and10 (a,b,c,d,e,f,g,h,i,j : bit) return bit is
begin
    return a and b and c and d and e and f and g and h
    and i and j;
end;
```

The resulting package declaration would then use the function in the body and the function header in the package header thus:

```vhdl
package new_functions is
    function and10 (a,b,c,d,e,f,g,h,i,j : bit) return bit;
end;
package body new_functions is
    function and10 (a,b,c,d,e,f,g,h,i,j : bit) return bit is
    begin
        return a and b and c and d and e \ 
        and f and g and h and i and j;
    end;
end;
```

**Components**

While procedures, functions and packages are useful in including behavioral constructs generally, with VHDL being used in a hardware design context, often there is a need to encapsulate design blocks as a separate component that can be included in a design, usually higher in the system hierarchy. The method for doing this in VHDL is called a COMPONENT. Caution needs to be exercised with components as the method of including components changed radically between VHDL 1987 and VHDL 1993, as such care needs to be taken to ensure that the correct language definitions are used consistently.

Components are a way of incorporating an existing VHDL entity and architecture into a new design without including the previously created model. The first step is to declare the component – in a similar way that functions need to be declared. For example, if an entity is called and4, and it has 4 inputs (a, b, c, d of type bit) and 1 output (q of type bit), then the component declaration would be of the form shown below:

```vhdl
component and4
    port ( a, b, c, d : in bit; q : out bit );
end component;
```
Then this component can be instantiated in a netlist form in the VHDL model architecture:

```vhdl
d1 : and4 port map ( a, b, c, d, q );
```

Note that in this case, there is no explicit mapping between port names and the signals in the current level of VHDL, the pins are mapped in the same order as defined in the component declaration. If each pin is to be defined independent of the order of the pins, then the explicit port map definition needs to be used:

```vhdl
d1: and4 port map ( a => a, b => b, c => c, d => d, q => q );
```

The final thing to note is that this is called the default binding. The binding is the link between the compiled architecture in the current library and the component being used. It is possible, for example, to use different architectures for different instantiated components using the following statement for a single specific device:

```vhdl
for d1 : and4 use entity work.and4(behaviour) port map (a,b,c,d,q);
```

or the following to specify a specific device for all the instantiated components:

```vhdl
for all : and4 use entity work.and4(behaviour) port map (a,b,c,d,q);
```

## Procedures

Procedures are similar to functions, except that they have more flexibility in the parameters, in that the direction can be in, out or inout. This is useful in comparison to functions where there is generally only a single output (although it may be an array) and avoids the need to create a record structure to manage the return value. Although procedures are useful, they should be used only for small specific functions. Components should be used to partition the design, not procedures, and this is especially true in FPGA design, as the injudicious use of procedures can lead to bloated and inefficient implementations, although the VHDL description can be very compact. A simple procedure to execute a full adder could be of the form:

```vhdl
procedure full_adder (a,b : in bit; sum, carry : out bit) is
begin
    sum := a xor b;
    carry := a and b;
end;
```
Notice that the syntax is the same as that for variables (NOT signals), and that multiple outputs are defined without the need for a return statement.

Debugging models

Assertions

Assertions are used to check if certain conditions have been met in the model and are extremely useful in debugging models. Some examples:

```vhdl
assert value <= max_value
  report "Value too large";
assert clock_width >= 100 ns
  report "clock width too small"
  severity failure;
```

Basic data types

Basic types

VHDL has the following standard types defined as built in data types:

- BIT
- BOOLEAN
- BIT_VECTOR
- INTEGER
- REAL

Data type: BIT

The BIT data type is the simple logic type built into VHDL. The type can have two legal values ‘0’ or ‘1’. The elements defined as of type BIT can have the standard VHDL built in logic functions applied to them. Examples of signal and variable declarations of type BIT follow:

```vhdl
signal ina : bit;
variable inb : bit := '0';
ina <= inb and inc;
ind <= '1' after 10 ns;
```
Data type: Boolean

The Boolean data type is primarily used for decision-making, so the test value for ‘if’ statements is a Boolean type. The elements defined as of type Boolean can have the standard VHDL built-in logic functions applied to them. Examples of signal and variable declarations of type Boolean follow:

```vhdl
signal test1 : Boolean;
variable test2 : Boolean := FALSE;
```

Data type: integer

The basic numeric type in VHDL is the integer and is defined as an integer in the range $-2^{31}$ to $2^{31}$. There are obviously implications for synthesis in the definition of integers in any VHDL model, particularly the effective number of bits, and so it is quite common to use a specified range of integer to constrain the values of the signals or variables to within physical bounds. Examples of integer usage follow:

```vhdl
signal int1 : integer;
variable int2 : integer := 124;
```

There are two subtypes (new types based on the fundamental type) derived from the integer type which are integer in nature, but simply define a different range of values.

**Integer subtypes: natural**

The natural subtype is used to define all integers greater than and equal to zero. They are actually defined with respect to the high value of the integer range as follows:

```vhdl
natural values : 0 to integer'high
```

**Integer subtypes: positive**

The positive subtype is used to define all integers greater than and equal to one. They are actually defined with respect to the high value of the integer range as follows:

```vhdl
positive values : 1 to integer'high
```

Data type: character

In addition to the numeric types inherent in VHDL, there are also the complete set of ASCII characters available for designers. There is no automatic conversion between characters and a
numeric value \textit{per se}; however, there is an implied ordering of the characters defined in the VHDL standard (IEEE Std 1076-1993). The characters can be defined as individual characters or arrays of characters to create strings. The best way to consider characters is an enumerated type.

**Data type: real**

Floating point numbers are used in VHDL to define real numbers and the predefined floating point type in VHDL is called real. This defines a floating point number in the range $-1.0 \times 10^{38}$ to $+10^{38}$. This is an important issue for many FPGA designs, as most commercial synthesis products do not support real numbers – precisely because they are floating point. In practice, it is necessary to use integer or fixed point numbers which can be directly and simply synthesized into hardware. An example of defining real signals or variables is shown below:

```vhdl
signal realno : real;
variable realno : real := 123.456;
```

**Data type: time**

Time values are defined using the special time type. These not only include the time value, but also the unit – separated by a space. The basic range of the time type value is between $-2^{31}$ and $2^{31}$, and the basic unit of time is defined as the femto-second (fs). Each subsequent time unit is derived from this basic unit of the fs as shown below:

```vhdl
ps = 1000 fs;
ns = 1000 ps;
us = 1000 ns;
ms = 1000 us;
min = 60 sec;
hr = 60 min;
```

Examples of time definitions are shown below:

```vhdl
delay : time := 10 ns;
wait for 20 us;
y <= x after 10 ms;
z <= y after delay;
```

**Summary**

This chapter provides a very brief introduction to VHDL and is certainly not a comprehensive reference. It enables the reader,
hopefully, to have enough knowledge to understand the syntax of the examples in this book. The author strongly recommends that anyone serious about design with VHDL should also obtain a detailed and comprehensive reference book on VHDL, such as Zwolinski (a useful introduction to digital design with VHDL – a common student textbook) or Ashenden (a more heavy duty VHDL reference that is perhaps more comprehensive, but less easy for a beginner to VHDL).