The overall goal of any hardware design is to ensure that the design meets the requirements of the design specification. In order to measure this it is indeed the case we need to not only simulate the design representation in a hardware description language (such as VHDL), but also to ensure that whatever tests we undertake are appropriate and demonstrate that the specification has been met.

The way that designers can test their designs in a simulator is by creating a ‘test bench’. This is directly analogous to a real experimental test bench in the sense that stimuli are defined, and the responses of the circuit measured to ensure that they meet the specification.

In practice, the test bench is simply a VHDL model that generates the required stimuli and checks the responses. This can be in such a way that the designer can view the waveforms and manually check them, or by using VHDL constructs to check the design responses automatically.

The goals of any test bench are twofold. The first is primarily to ensure that correct operation is achieved. This is essentially a ‘functional’ test. The second goal is to ensure that a synthesised design still meets the specification (particularly with a view to timing errors).
Simple test bench: instantiating components

Consider a simple combinatorial VHDL model given below:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity cct is
    port ( in0, in1 : in std_logic;
           out1 : out std_logic);
end;
architecture simple of cct is
begin
    out1 <= in0 AND in1;
end;
```

This simple model is clearly a two input AND gate, and to test the operation of the component we need to do several things.

First, we must include the component in a new VHDL design. So we need to create a basic test bench. The listing below shows how a basic entity (with no connections) is created, and then the architecture contains both the component declaration and the signals to test the design.

```vhdl
-- library declarations
library ieee;
use ieee.std_logic_1164.all;
-- empty entity declaration
entity test is
end;
-- test bench architecture
architecture testbench of test is
    -- component declaration
    component cct
        port ( in0, in1 : in std_logic;
               out1 : out std_logic);
    end component;
    -- test bench signal declarations
    signal in0, in1, out1 : std_logic;
    -- architecture body
    Begin
        -- declare the Circuit Under Test (CUT)
        CUT : cct port map ( in0, in1, out1 );
    end;
```

This test bench will compile in a VHDL simulator, but is not particularly useful as there are no definitions of the input stimuli (signals in0 and in1) that will exercise the Circuit Under Test (CUT).
If we wish to add stimuli to our test bench we have some significant advantages over our design VHDL – the most appealing is that we generally don’t need to adhere to any design rules or even make the code synthesisable. Test bench code is generally designed to be ‘off chip’ and therefore we can make the code as abstract or behavioral as we like and it will still be fit for purpose. We can use wait statements, file read and write, assertions and other non-synthesisable code options.

Adding stimuli
In order to add a basic set of stimuli to our test bench, we could simply define the values of the input signals in0 and in1 with a simple signal assignment:

```vhdl
begin
  CUT : cct port map ( in0, in1, out1 );
  in0 <= '0';
  in1 <= '1';
end;
```

Clearly this is not very complex or dynamic test bench, so to add a sequence of events we can modify the signal assignments to include numerous value, time pairs defining a sequence of values.

```vhdl
begin
  CUT : cct port map ( in0, in1, out1 );
  in0 <= '0' after 0 ns, '1' after 10 ns, '0' after 20 ns;
  in1 <= '0' after 0 ns, '1' after 15 ns, '0' after 25 ns;
end;
```

While this method is useful for small circuits, clearly for more complex realistic designs it is of limited value. Another approach is to define a constant array of values that allow a number of tests to be carried out with a relatively simple test bench and applying a different set of stimuli and responses in turn.

For example, we can exhaustively test our simple two input logic design using a set of data in a `record`. A VHDL record is simply a collection of types grouped together defined as a new type.

```vhdl
type testdata is record
  in0 : std_logic;
  in1 : std_logic;
end;
```
With a new composite type, such as a record, we can then create an array, just as in any standard VHDL type. This requires another type declaration, of the array type itself.

```vhdl
type data_array is array (natural range <> ) of data_array
```

With these two new types we can simply declare a constant (of type data_array) that is an array of record values (of type testdata) that fully describe the data set to be used to test the design. Notice that the type data_array does not have a default range, but that this is defined by the declaration in this particular test bench.

```vhdl
constant test_data : data_array := ( ('0', '0'), ('0', '1'), ('1', '0'), ('1', '1') );
```

The beauty of this approach is that we can change from a system that requires every test stimulus to be defined explicitly, to one where a generic test data process will read values from pre-defined arrays of data. In the simple test example presented here, an example process to apply each set of test data in turn could be implemented as follows:

```vhdl
process
begin
  for i in test_data'range loop
    in0 <= test_data(i).in0;
    in1 <= test_data(i).in1;
    wait for 100 ns;
  end loop
  wait;
end process;
```

There are several interesting aspects to this piece of test bench VHDL. The first is that we can use behavioral VHDL (wait for 100 ns) as we are not constrained to synthesize this to hardware. Secondly, by using the range operator, the test bench becomes unconstrained by the size of the data set. Finally, the individual record elements are accessed using the hierarchical construct test_data(i).in0 or test_data(i).in1, respectively.

**Libraries**

**Introduction**

VHDL as a language on its own is actually very limited in the breadth of the data types and primitive models available. As a result, libraries are required to facilitate design re-use and standard data
types for model exchange, re-use and synthesis. The primary library for standard VHDL design is the IEEE library. Within the IEEE Design Automation Standards Committee (DASC), various committees have developed libraries, packages and extensions to standard VHDL. Some of these are listed below:

- IEEE Std 1076 Standard VHDL Language
- IEEE Std 1076.1 Standard VHDL Analog and Mixed-Signal Extensions (VHDL-AMS)
- IEEE Std 1076.1.1 Standard VHDL Analog and Mixed-Signal Extensions – Packages for Multiple Energy Domain Support
- IEEE Std 1076.4 Standard VITAL ASIC (Application Specific Integrated Circuit) Modeling Specification (VITAL)
- IEEE Std 1076.6 Standard for VHDL Register Transfer Level (RTL) Synthesis (SIWG)
- IEEE Std 1076.2 IEEE Standard VHDL Mathematical Packages (math)
- IEEE Std 1076.3 Standard VHDL Synthesis Packages (vhdl synth)
- IEEE Std 1164 Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)

Each of these ‘working groups’ are volunteers who come from a combination of academia, EDA industry and user communities, and collaborate to produce the IEEE Standards (usually revised every 4 years).

**Using libraries**

In order to use a library, first the library must be declared:

```vhdl
library ieee;
```

Within each library a number of VHDL packages are defined, that allow specific data types or functions to be employed in the design. For example, in digital systems design, we require logic data types, and these are not defined in the basic VHDL standard (1076). Standard VHDL defines integer, boolean and bit types, but not a standard logic definition. This is obviously required for digital design and an appropriate IEEE standard was developed for this purpose – IEEE Std 1164. It is important to note that IEEE Std 1164 is
NOT a subset of VHDL (IEEE 1076), but is defined for hardware description languages in general.

**Std_logic libraries**

There are a number of std_logic libraries available in the IEEE library and these are:

- std_logic_1164
- std_logic_arith
- std_logic_unsigned
- std_logic_signed
- std_logic_entities
- std_logic_components
- std_logic_misc
- std_logic_textio

In order to use a particular element of a package in a design, the user is required to declare their use of a package using the USE command. For example, to use the standard IEEE logic library, the use needs to add a declaration after the library declaration as follows:

```verbatim
library ieee;
use ieee.std_logic_1164.all;
```

The std_logic_1164 package is particularly important for most digital design, especially for Field Programmable Gate Array (FPGA), because it defines the standard logic types used by ALL the commercially available simulation and synthesis software tools, and is included as a standard library. It incorporates not only the definition of the standard logic types, but also conversion functions (to and from the standard logic types) and also manages the conversion between signed, unsigned and logic array variables.

**Std_logic type definition**

As it is such an important type, the std_logic type is described in this section. The type has the following definition:

- ‘U’: uninitialized; this signal hasn’t been set yet
- ‘X’: unknown; impossible to determine this value/result
- ‘0’: logic 0
• ‘1’: logic 1
• ‘Z’: High Impedance
• ‘W’: Weak signal, can’t tell if it should be 0 or 1
• ‘L’: Weak signal that should probably go to 0
• ‘H’: Weak signal that should probably go to 1
• ‘-’: Don’t care

These definitions allow resolution of logic signals in digital designs in a standard manner that is predictable and repeatable across software tools and platforms. The operations that can be carried out on the basic std_logic data types are the standard built in VHDL logic functions:

• and
• nand
• or
• nor
• xor
• xnor
• not

An example of the use of the std_logic library would be to define a simple logic gate – in this case a three input nand gate:

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

entity nand3 is
    port ( in0, in1, in2 : in std_logic;
           out1 : out std_logic ) ;
end;

architecture simple of nand3 is
begin
    out1 <= in0 nand in1 nand in2;
end;
```

**Synthesis**

**Design flow for synthesis**

The basic HDL design flow is shown in Figure 7.
As can be seen from this figure, Synthesis is the key stage between high-level design and the physical place and route which is the final product of the design flow. There are several different types of synthesis ranging from behavioral, to RTL and finally physical synthesis.

Behavioral synthesis is the mechanism by which high level abstract models are synthesized to an intermediate model that is physically realizable. Behavioral models can be written in VHDL that are not directly synthesizable and so care must be taken with high level models to ensure that this can take place, in fact. There are limited tools that can synthesize behavioral VHDL and these include the Behavioral Compiler from Synopsys, Inc and MOODS, a research synthesis platform from the University of Southampton.

RTL Synthesis is what most designers call synthesis, and is the mechanism whereby a direct translation of structural and register level VHDL can be synthesized to individual gates targeted at a specific FPGA platform. At this stage, detailed timing analysis can be carried out and an estimate of power consumption obtained. There are numerous commercial synthesis software packages including Design Compiler (Synopsys), Leonardo Spectrum (Mentor Graphics) and Synplify (Synplicity) – but this is not an exhaustive list – there are numerous offerings available at a variety of prices.

Physical synthesis is the last stage in a synthesis design flow and is where the individual gates are placed (using a ‘floorplan’) and routed on the specific FPGA platform.
Synthesis issues

Synthesis basically transforms program-like VHDL into a true hardware design (netlist). It requires a set of inputs, a VHDL description, timing constraints (when outputs need to be ready, when inputs will be ready, data to estimate wire delay), a technology to map to (list of available blocks and their size/timing information) and information about design priorities (area vs. speed)

For big designs, the VHDL will typically be broken into modules and then synthesized separately. 10 K gates per module was a reasonable size in the 1990s, however tools can handle a lot more now.

RTL design flow

RTL VHDL is the input to most standard synthesis software tools. The VHDL must be written in a form that contains Registers, State Machines (FSM) and combinational logic functions. The synthesis software translates these blocks and functions into gates and library cells from the FPGA library. The RTL design flow is shown in Figure 8, in more detail than the overall HDL design flow. Using
RTL VHDL restricts the scope of the designer as it precludes algorithmic design – as we shall see later. This approach forces the designer to think at quite a low level – making the resulting code sometimes verbose and cumbersome. It also forces structural decisions early in the design process – restrictive and not always advisable, or helpful.

The Design process starts from RTL VHDL:

- Simulation (RTL) – is needed to develop a test bench (VHDL).
- Synthesis (RTL) – targeted at a standard FPGA platform.
- Timing simulation (Structural) – simulate to check timing.
- Place and route using standard tools (e.g. Xilinx Design Manager).

Although there are a variety of software tools available for synthesis (such as Leonardo Spectrum or Synplify), they all have generally similar approaches and design flows.

**Physical design flow**

Synthesis generates a netlist of devices plus interconnections. The ‘place and route’ software figures out where the devices go and how to connect them. The results not as good as you’d like; a 40 to 60 per cent utilization of devices and wires is typical. The designer can trade off run time against greater utilization to some degree, but there are serious limits. Typically the FPGA vendor will provide a software toolkit (such as the Xilinx Design Navigator or Altera’s Quartus tools) that manages the steps involved in physical design.

Regardless of the particular physical synthesis flow chosen, the steps required to translate the VHDL or EDIF output from an RTL synthesis software program into a physically downloadable bit file are essentially the same and are listed below:

1. Translate
2. Map
3. Place
4. Route
5. Generate accurate timing models and reports
6. Create binary files for download to device
Place and route

There are two main techniques to place and route in current commercial software which are recursive cut and simulated annealing.

Recursive cut

In a recursive cut algorithm, we divide the netlist into two halves, move devices between halves to minimize the number of wires that cross cut (while keeping the number of devices in each half the same). This is repeated to get smaller and smaller blocks.

Timing analysis

Static timing analysis is the most commonly-used approach. In static timing analysis, we calculate the delay from each input to each output of all devices. The delays are added up along each path through circuit to get the critical path through the design and hence the fastest design speed.

This works as long as there are no cycles in the circuit, however in these cases the analysis becomes less easy. Design software allows you to break cycles at registers to handle feedback if this is the case.

As in any timing analysis, the designer can trade off some accuracy for run time. Digital simulation software such as Modelism or Verilog will give fast results, but will use approximate models of timing, whereas analog simulation tools like SPICE will give more accurate numbers, but take much longer to run.

Design pitfalls

The most common mistake that inexperienced designers make is simply making things too complex. The best approach to successful design is to keep the design elements simple, and the easiest way to manage that is efficient use of hierarchy.

The second mistake that is closely related to design complexity is not testing enough. It is vital to ensure that all aspects of the design are adequately tested. This means not only carrying out basic functional testing, but also systematic testing, and checking for redundant states and potential error states.

Another common pitfall is to use multiple clocks unnecessarily. Multiple clocks can create timing related bugs that are transient or
hardware dependent. They can also occur in hardware and yet be missed by simulation.

**VHDL issues for FPGA design**

**Initialization**

Any default values of signals and variables are ignored. This means that you must ensure that synchronous (or asynchronous) sets and resets must be used on all flip-flops to ensure a stable starting condition. Remember that synthesis tools are basically stupid and follows a basic set of rules that may not always result in the hardware that you expect.

**Floating point numbers and operations**

Data types using floating point are currently not supported by synthesis software tools. They generally require 32 bits and the requisite hardware is just too large for most FPGA and ASIC platforms.

**Summary**

This chapter has introduced the practical aspect of developing test benches and validating VHDL models using simulation. This is an often overlooked skill in VHDL (or any hardware description language) and is vital to ensuring correct behavior of the final implemented design. We have also introduced the concept of design synthesis and highlighted the problem of not only ensuring that a design simulates correctly, but also how we can make sure that the design will synthesize to the target technology and still operate correctly with practical delays and parasitics. Finally, we have raised some of the practical implementation issues and potential problems that can occur with real designs, and these will be discussed in more detail in Part 4 of this book.

An important concept useful to define here is the difference between validation and verification. The terms are often confused leading to problems in the final design and meeting a specification. Validation is the task of ensuring that the design is ‘doing the right thing’. If the specification asks for a low pass filter, then we must implement a low pass filter to have a valid design. We can even be more specific and state that the design must perform
within a constraint. Verification, on the other hand, is much more specific and can be stated as ‘doing the right thing right’. In other words, verification is ensuring that not only does our design do what is required functionally, but in addition it must meet ALL the criteria defined by the specification, preferably with some headroom to ensure that the design will operate to the specification under all possible operating conditions.