1. Introduction

VHDL is a language for describing digital electronic systems. It arose out of the United States Government's Very High Speed Integrated Circuits (VHSIC) program, initiated in 1980. In the course of this program, it became clear that there was a need for a standard language for describing the structure and function of integrated circuits (ICs). Hence the VHSIC Hardware Description Language (VHDL) was developed, and subsequently adopted as a standard by the Institute of Electrical and Electronic Engineers (IEEE) in the US.

VHDL is designed to fill a number of needs in the design process. Firstly, it allows description of the structure of a design, that is how it is decomposed into sub-designs, and how those sub-designs are interconnected. Secondly, it allows the specification of the function of designs using familiar programming language forms. Thirdly, as a result, it allows a design to be simulated before being manufactured, so that designers can quickly compare alternatives and test for correctness without the delay and expense of hardware prototyping.

The purpose of this booklet is to give you a quick introduction to VHDL. This is done by informally describing the facilities provided by the language, and using examples to illustrate them. This booklet does not fully describe every aspect of the language. For such fine details, you should consult the *IEEE Standard VHDL Language Reference Manual*. However, be warned: the standard is like a legal document, and is very difficult to read unless you are already familiar with the language. This booklet does cover enough of the language for substantial model writing. It assumes you know how to write computer programs using a conventional programming language such as Pascal, C or Ada.

The remaining chapters of this booklet describe the various aspects of VHDL in a bottom-up manner. Chapter 2 describes the facilities of VHDL which most resemble normal sequential programming languages. These include data types, variables, expressions, sequential statements and subprograms. Chapter 3 then examines the facilities for describing the structure of a module and how it it decomposed into sub-modules. Chapter 4 covers aspects of VHDL that integrate the programming language features with a discrete event timing model to allow simulation of behaviour. Chapter 5 is a key chapter that shows how all these facilities are combined to form a complete model of a system. Then Chapter 6 is a pot-pourri of more advanced features which you may find useful for modeling more complex systems.

Throughout this booklet, the syntax of language features is presented in Backus-Naur Form (BNF). The syntax specifications are drawn from the IEEE VHDL Standard. Concrete examples are also given to illustrate the language features. In some cases, some alternatives are omitted from BNF

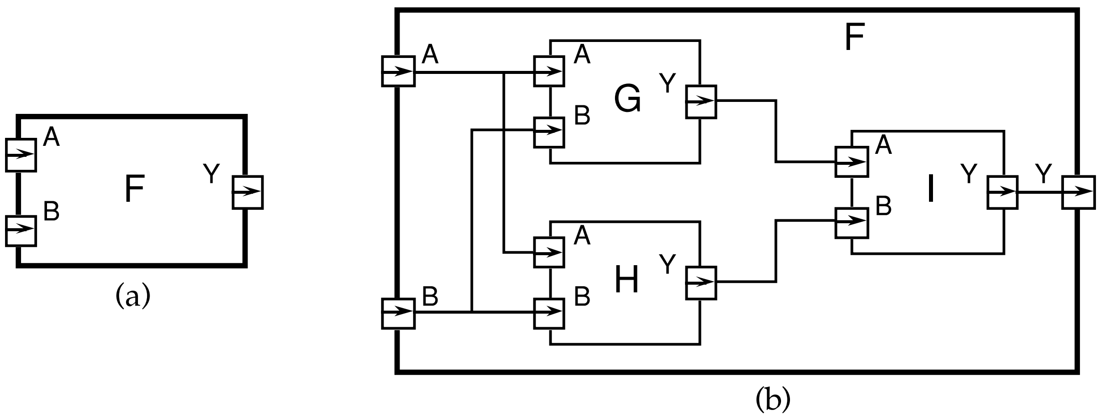

Figure 1-1. Example of a structural description.

productions where they are not directly relevant to the context. For this reason, the full syntax is included in Appendix A, and should be consulted as a reference.

1.1. Describing Structure

A digital electronic system can be described as a module with inputs and/or outputs. The electrical values on the outputs are some function of the values on the inputs. Figure 1-1(a) shows an example of this view of a digital system. The module F has two inputs, A and B, and an output Y. Using VHDL terminology, we call the module F a design *entity*, and the inputs and outputs are called *ports*.

One way of describing the function of a module is to describe how it is composed of sub-modules. Each of the sub-modules is an *instance* of some entity, and the ports of the instances are connected using *signals*. Figure 1-1(b) shows how the entity F might be composed of instances of entities G, H and I. This kind of description is called a *structural* description. Note that each of the entities G, H and I might also have a structural description.

1.2. Describing Behaviour

In many cases, it is not appropriate to describe a module structurally. One such case is a module which is at the bottom of the hierarchy of some other structural description. For example, if you are designing a system using IC packages bought from an IC shop, you do not need to describe the internal structure of an IC. In such cases, a description of the function performed by the module is required, without reference to its actual internal structure. Such a description is called a *functional* or *behavioural* description.

To illustrate this, suppose that the function of the entity F in Figure 1-1(a) is the exclusive-or function. Then a behavioural description of F could be the Boolean function

$$Y = \overline{A} . B + A . \overline{B}$$

More complex behaviours cannot be described purely as a function of inputs. In systems with feedback, the outputs are also a function of time. VHDL solves this problem by allowing description of behaviour in the form of an executable program. Chapters 2 and 4 describe the programming language facilities.

1.3. Discrete Event Time Model

Once the structure and behaviour of a module have been specified, it is possible to simulate the module by executing its bevioural description. This is done by simulating the passage of time in discrete steps. At some simulation time, a module input may be stimulated by changing the value on an input port. The module reacts by running the code of its behavioural description and scheduling new values to be placed on the signals connected to its output ports at some later simulated time. This is called scheduling a *transaction* on that signal. If the new value is different from the previous value on the signal, an *event* occurs, and other modules with input ports connected to the signal may be activated.

The simulation starts with an *initialisation phase*, and then proceeds by repeating a two-stage *simulation cycle*. In the initialisation phase, all signals are given initial values, the simulation time is set to zero, and each module's behaviour program is executed. This usually results in transactions being scheduled on output signals for some later time.

In the first stage of a simulation cycle, the simulated time is advanced to the earliest time at which a transaction has been scheduled. All transactions scheduled for that time are executed, and this may cause events to occur on some signals.

In the second stage, all modules which react to events occurring in the first stage have their behaviour program executed. These programs will usually schedule further transactions on their output signals. When all of the behaviour programs have finished executing, the simulation cycle repeats. If there are no more scheduled transactions, the whole simulation is completed.

The purpose of the simulation is to gather information about the changes in system state over time. This can be done by running the simulation under the control of a *simulation monitor*. The monitor allows signals and other state information to be viewed or stored in a trace file for later analysis. It may also allow interactive stepping of the simulation process, much like an interactive program debugger.

1.4. A Quick Example

In this section we will look at a small example of a VHDL description of a two-bit counter to give you a feel for the language and how it is used. We start the description of an entity by specifying its external interface, which includes a description of its ports. So the counter might be defined as:

```
entity count2 is
    generic (prop_delay : Time := 10 ns);
    port (clock : in bit;
          q1, q0 : out bit);
end count2;
```

This specifies that the entity count2 has one input and two outputs, all of which are bit values, that is, they can take on the values '0' or '1'. It also defines a generic constant called prop_delay which can be used to control the operation of the entity (in this case its propagation delay). If no value is

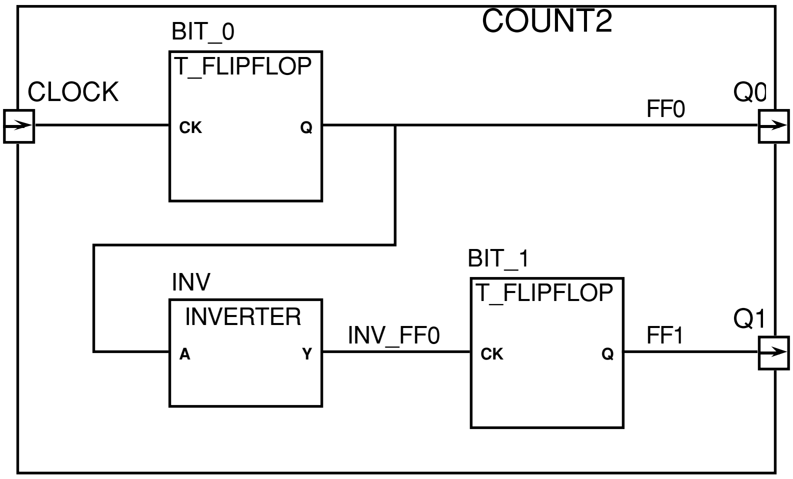

Figure 1-2. Structure of count2.

explicitly given for this value when the entity is used in a design, the default value of 10 ns will be used.

An implementation of the entity is described in an architecture body. There may be more than one architecture body corresponding to a single entity specification, each of which describes a different view of the entity. For example, a behavioural description of the counter could be written as:

```
architecture behaviour of count2 is
begin
    count_up: process (clock)
        variable count_value : natural := 0;
    begin
        if clock = '1' then
            count_value := (count_value + 1) mod 4;
            q0 <= bit'val(count_value mod 2) after prop_delay;
            q1 <= bit'val(count_value / 2) after prop_delay;
        end if;
    end process count_up;
end behaviour;
```

In this description of the counter, the behaviour is implemented by a process called count_up, which is sensitive to the input clock. A process is a body of code which is executed whenever any of the signals it is sensitive to changes value. This process has a variable called count_value to store the current state of the counter. The variable is initialized to zero at the start of simulation, and retains its value between activations of the process. When the clock input changes from '0' to '1', the state variable is incremented, and transactions are scheduled on the two output ports based on the new value. The assignments use the generic constant prop_delay to determine how long after the clock change the transaction should be scheduled. When control reaches the end of the process body, the process is suspended until another change occurs on clock.

The two-bit counter might also be described as a circuit composed of two T-flip-flops and an inverter, as shown in Figure 1-2. This can be written in VHDL as:

```
architecture structure of count2 is
    component t_flipflop
        port (ck : in bit;  q : out bit);
    end  component;
    component inverter
        port (a : in bit;  y : out bit);
    end  component;
    signal ff0, ff1, inv_ff0 : bit;
begin
    bit_0 : t_flipflop port map (ck => clock, q => ff0);
    inv : inverter port map (a => ff0, y => inv_ff0);
    bit_1 : t_flipflop port map (ck => inv_ff0, q => ff1);
    q0 <= ff0;
    q1 <= ff1;
end  structure;
```

In this architecture, two component types are declared, t_flipflop and inverter, and three internal signals are declared. Each of the components is then instantiated, and the ports of the instances are mapped onto signals and ports of the entity. For example, bit_0 is an instance of the t_flipflop component, with its ck port connected to the clock port of the count2 entity, and its q port connected to the internal signal ff0. The last two signal assignments update the entity ports whenever the values on the internal signals change.