



The VHDL Cookbook

First Edition



Peter J. Ashenden

The VHDL Cookbook

First Edition

July, 1990

Peter J. Ashenden

Dept. Computer Science
University of Adelaide
South Australia

© 1990, Peter J. Ashenden

Contents

1.	Introduction.....	1-1
1.1.	Describing Structure	1-2
1.2.	Describing Behaviour	1-2
1.3.	Discrete Event Time Model.....	1-3
1.4.	A Quick Example.....	1-3
2.	VHDL is Like a Programming Language	2-1
2.1.	Lexical Elements	2-1
2.1.1.	Comments	2-1
2.1.2.	Identifiers.....	2-1
2.1.3.	Numbers	2-1
2.1.4.	Characters.....	2-2
2.1.5.	Strings	2-2
2.1.6.	Bit Strings.....	2-2
2.2.	Data Types and Objects	2-2
2.2.1.	Integer Types	2-3
2.2.2.	Physical Types.....	2-3
2.2.3.	Floating Point Types.....	2-4
2.2.4.	Enumeration Types.....	2-4
2.2.5.	Arrays.....	2-5
2.2.6.	Records	2-7
2.2.7.	Subtypes	2-7
2.2.8.	Object Declarations	2-8
2.2.9.	Attributes	2-8
2.3.	Expressions and Operators	2-9
2.4.	Sequential Statements	2-10
2.4.1.	Variable Assignment.....	2-10
2.4.2.	If Statement	2-11
2.4.3.	Case Statement.....	2-11
2.4.4.	Loop Statements	2-12
2.4.5.	Null Statement	2-13
2.4.6.	Assertions	2-13
2.5.	Subprograms and Packages	2-13
2.5.1.	Procedures and Functions	2-14
2.5.2.	Overloading	2-16
2.5.3.	Package and Package Body Declarations	2-17
2.5.4.	Package Use and Name Visibility	2-18

Contents (cont'd)

3.	VHDL Describes Structure	3-1
3.1.	Entity Declarations	3-1
3.2.	Architecture Declarations	3-3
3.2.1.	Signal Declarations	3-3
3.2.2.	Blocks	3-4
3.2.3.	Component Declarations	3-5
3.2.4.	Component Instantiation	3-6
4.	VHDL Describes Behaviour	4-1
4.1.	Signal Assignment	4-1
4.2.	Processes and the Wait Statement	4-2
4.3.	Concurrent Signal Assignment Statements	4-4
4.3.1.	Conditional Signal Assignment	4-5
4.3.2.	Selected Signal Assignment	4-6
5.	Model Organisation	5-1
5.1.	Design Units and Libraries	5-1
5.2.	Configurations	5-2
5.3.	Complete Design Example	5-5
6.	Advanced VHDL	6-1
6.1.	Signal Resolution and Buses	6-1
6.2.	Null Transactions	6-2
6.3.	Generate Statements	6-2
6.4.	Concurrent Assertions and Procedure Calls	6-3
6.5.	Entity Statements	6-4
7.	Sample Models: The DP32 Processor	7-1
7.1.	Instruction Set Architecture	7-1
7.2.	Bus Architecture	7-4
7.3.	Types and Entity	7-6
7.4.	Behavioural Description	7-9
7.5.	Test Bench	7-18
7.6.	Register Transfer Architecture	7-24
7.6.1.	Multiplexor	7-25
7.6.2.	Transparent Latch	7-25
7.6.3.	Buffer	7-26
7.6.4.	Sign Extending Buffer	7-28
7.6.5.	Latching Buffer	7-28
7.6.6.	Program Counter Register	7-28
7.6.7.	Register File	7-29

Contents (cont'd)

7.6.8. Arithmetic & Logic Unit.....7-30

7.6.9. Condition Code Comparator.....7-34

7.6.10. Structural Architecture of the DP32.....7-34

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. Chapter2 describes the facilities of VHDL which most resemble normal sequential programming languages. These include data types, variables, expressions, sequential statements and subprograms. Chapter3 then examines the facilities for describing the structure of a module and how it is decomposed into sub-modules. Chapter4 covers aspects of VHDL that integrate the programming language features with a discrete event timing model to allow simulation of behaviour. Chapter5 is a key chapter that shows how all these facilities are combined to form a complete model of a system. Then Chapter6 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

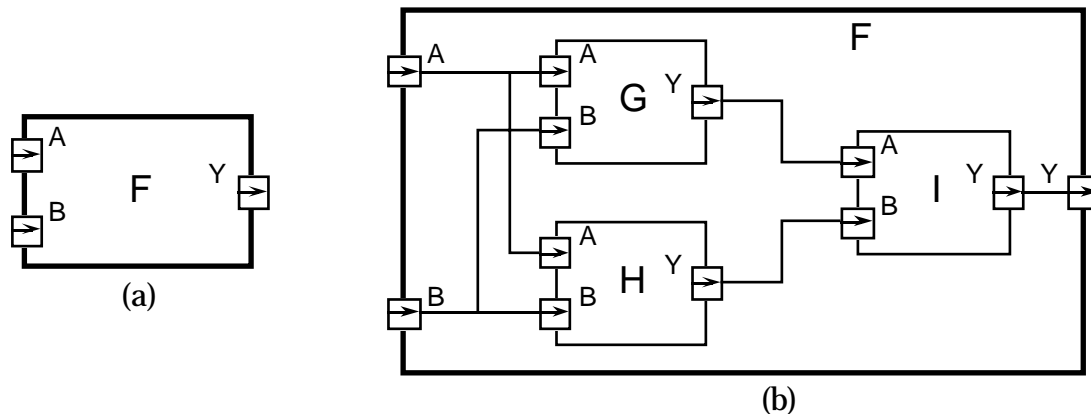


Figure 1-1. Example of a structural description.

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

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

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 behavioural 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

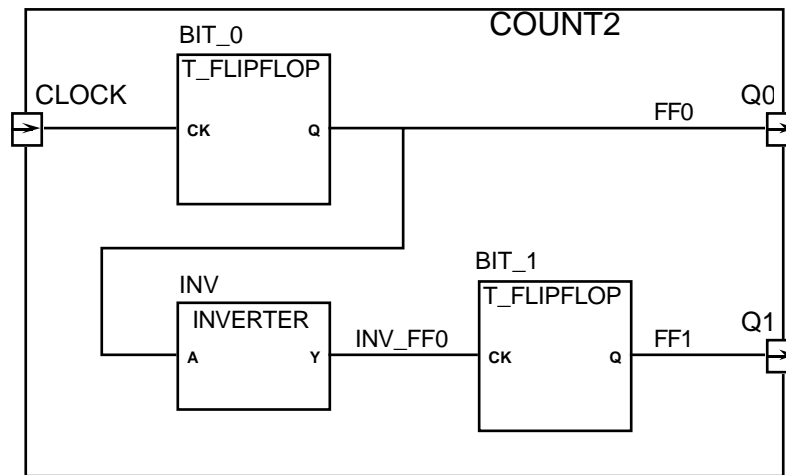


Figure1-2. Structure of count2.

explicitly given for this value when the entity is used in a design, the default value of 10ns 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 Figure1-2. 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.

2. VHDL is Like a Programming Language

As mentioned in Section 1.2, the behaviour of a module may be described in programming language form. This chapter describes the facilities in VHDL which are drawn from the familiar programming language repertoire. If you are familiar with the Ada programming language, you will notice the similarity with that language. This is both a convenience and a nuisance. The convenience is that you don't have much to learn to use these VHDL facilities. The problem is that the facilities are not as comprehensive as those of Ada, though they are certainly adequate for most modeling purposes.

2.1. Lexical Elements

2.1.1. Comments

Comments in VHDL start with two adjacent hyphens ('--') and extend to the end of the line. They have no part in the meaning of a VHDL description.

2.1.2. Identifiers

Identifiers in VHDL are used as reserved words and as programmer defined names. They must conform to the rule:

```
identifier ::= letter { [ underline ] letter_or_digit }
```

Note that case of letters is not considered significant, so the identifiers cat and Cat are the same. Underline characters in identifiers are significant, so This_Name and ThisName are different identifiers.

2.1.3. Numbers

Literal numbers may be expressed either in decimal or in a base between two and sixteen. If the literal includes a point, it represents a real number, otherwise it represents an integer. Decimal literals are defined by:

```
decimal_literal ::= integer [ . integer ] [ exponent ]
```

```
integer ::= digit { [ underline ] digit }
```

```
exponent ::= E [ + ] integer | E - integer
```

Some examples are:

```
0      1      123_456_789      987E6      -- integer literals
```

```
0.0    0.5    2.718_28          12.4E-9     -- real literals
```

Based literal numbers are defined by:

```
based_literal ::= base # based_integer [ . based_integer ] # [ exponent ]
```

```
base ::= integer
```

```
based_integer ::= extended_digit { [ underline ] extended_digit }
```

```
extended_digit ::= digit | letter
```

The base and the exponent are expressed in decimal. The exponent indicates the power of the base by which the literal is multiplied. The letters A to F (upper or lower case) are used as extended digits to represent 10 to 15. Some examples:

```
2#1100_0100#    16#C4#    4#301#E1    -- the integer 196
2#1.1111_1111_111#E+11  16#F.FF#E2    -- the real number 4095.0
```

2.1.4. Characters

Literal characters are formed by enclosing an ASCII character in single-quote marks. For example:

```
'A'    '*'    "'"    ' '
```

2.1.5. Strings

Literal strings of characters are formed by enclosing the characters in double-quote marks. To include a double-quote mark itself in a string, a pair of double-quote marks must be put together. A string can be used as a value for an object which is an array of characters. Examples of strings:

```
"A string"
""
"A string in a string: ""A string""." -- empty string
-- contains quote marks
```

2.1.6. Bit Strings

VHDL provides a convenient way of specifying literal values for arrays of type bit ('0's and '1's, see Section 2.2.5). The syntax is:

```
bit_string_literal ::= base_specifier " bit_value "
base_specifier ::= B | O | X
bit_value ::= extended_digit { [ underline ] extended_digit }
```

Base specifier B stands for binary, O for octal and X for hexadecimal. Some examples:

```
B"1010110"    -- length is 7
O"126"        -- length is 9, equivalent to B"001_010_110"
X"56"         -- length is 8, equivalent to B"0101_0110"
```

2.2. Data Types and Objects

VHDL provides a number of basic, or *scalar*, types, and a means of forming *composite* types. The scalar types include numbers, physical quantities, and enumerations (including enumerations of characters), and there are a number of standard predefined basic types. The composite types provided are arrays and records. VHDL also provides *access* types (pointers) and *files*, although these will not be fully described in this booklet.

A data type can be defined by a type declaration:

```
full_type_declaration ::= type identifier is type_definition ;
type_definition ::=
    scalar_type_definition
    | composite_type_definition
    | access_type_definition
    | file_type_definition
scalar_type_definition ::=
    enumeration_type_definition | integer_type_definition
    | floating_type_definition | physical_type_definition
```

```

composite_type_definition ::=
    array_type_definition
    | record_type_definition

```

Examples of different kinds of type declarations are given in the following sections.

2.2.1. Integer Types

An integer type is a range of integer values within a specified range. The syntax for specifying integer types is:

```

integer_type_definition ::= range_constraint
range_constraint ::= range range
range ::= simple_expression direction simple_expression
direction ::= to | downto

```

The expressions that specify the range must of course evaluate to integer numbers. Types declared with the keyword **to** are called *ascending* ranges, and those declared with the keyword **downto** are called *descending* ranges. The VHDL standard allows an implementation to restrict the range, but requires that it must at least allow the range -2147483647 to $+2147483647$.

Some examples of integer type declarations:

```

type byte_int is range 0 to 255;
type signed_word_int is range  $-32768$  to 32767;
type bit_index is range 31 downto 0;

```

There is a predefined integer type called `integer`. The range of this type is implementation defined, though it is guaranteed to include -2147483647 to $+2147483647$.

2.2.2. Physical Types

A physical type is a numeric type for representing some physical quantity, such as mass, length, time or voltage. The declaration of a physical type includes the specification of a base unit, and possibly a number of secondary units, being multiples of the base unit. The syntax for declaring physical types is:

```

physical_type_definition ::=
    range_constraint
    units
        base_unit_declaration
        { secondary_unit_declaration }
    end units
base_unit_declaration ::= identifier ;
secondary_unit_declaration ::= identifier = physical_literal ;
physical_literal ::= [ abstract_literal ] unit_name

```

Some examples of physical type declarations:

```

type length is range 0 to 1E9
  units
    um;
    mm = 1000 um;
    cm = 10 mm;
    m = 1000 mm;
    in = 25.4 mm;
    ft = 12 in;
    yd = 3 ft;
    rod = 198 in;
    chain = 22 yd;
    furlong = 10 chain;
  end units;

type resistance is range 0 to 1E8
  units
    ohms;
    kohms = 1000 ohms;
    Mohms = 1E6 ohms;
  end units;

```

The predefined physical type time is important in VHDL, as it is used extensively to specify delays in simulations. Its definition is:

```

type time is range implementation_defined
  units
    fs;
    ps = 1000 fs;
    ns = 1000 ps;
    us = 1000 ns;
    ms = 1000 us;
    sec = 1000 ms;
    min = 60 sec;
    hr = 60 min;
  end units;

```

To write a value of some physical type, you write the number followed by the unit. For example:

```
10 mm  1 rod  1200 ohm  23 ns
```

2.2.3. Floating Point Types

A floating point type is a discrete approximation to the set of real numbers in a specified range. The precision of the approximation is not defined by the VHDL language standard, but must be at least six decimal digits. The range must include at least $-1\text{E}38$ to $+1\text{E}38$. A floating point type is declared using the syntax:

```
floating_type_definition := range_constraint
```

Some examples are:

```

type signal_level is range -10.00 to +10.00;
type probability is range 0.0 to 1.0;

```

There is a predefined floating point type called real. The range of this type is implementation defined, though it is guaranteed to include $-1\text{E}38$ to $+1\text{E}38$.

2.2.4. Enumeration Types

An enumeration type is an ordered set of identifiers or characters. The identifiers and characters within a single enumeration type must be distinct, however they may be reused in several different enumeration types.

The syntax for declaring an enumeration type is:

```
enumeration_type_definition ::= ( enumeration_literal { , enumeration_literal } )
enumeration_literal ::= identifier | character_literal
```

Some examples are:

```
type logic_level is (unknown, low, undriven, high);
type alu_function is (disable, pass, add, subtract, multiply, divide);
type octal_digit is ('0', '1', '2', '3', '4', '5', '6', '7');
```

There are a number of predefined enumeration types, defined as follows:

```
type severity_level is (note, warning, error, failure);
type boolean is (false, true);
type bit is ('0', '1');
type character is (
    NUL,    SOH,    STX,    ETX,    EOT,    ENQ,    ACK,    BEL,
    BS,     HT,     LF,     VT,     FF,     CR,     SO,     SI,
    DLE,    DC1,    DC2,    DC3,    DC4,    NAK,    SYN,    ETB,
    CAN,    EM,     SUB,    ESC,    FSP,    GSP,    RSP,    USP,
    '\',    '\'',    '"',    '#',    '$',    '%',    '&',    '"',
    '(',    ')',    '*',    '+',    ',',    '-',    '.',    '/',
    '0',    '1',    '2',    '3',    '4',    '5',    '6',    '7',
    '8',    '9',    ':',    ';',    '<',    '>',    '=',    '?',
    '@',    'A',    'B',    'C',    'D',    'E',    'F',    'G',
    'H',    'I',    'J',    'K',    'L',    'M',    'N',    'O',
    'P',    'Q',    'R',    'S',    'T',    'U',    'V',    'W',
    'X',    'Y',    'Z',    '[',    '\',    ']',    '^',    '_',
    '`',    'a',    'b',    'c',    'd',    'e',    'f',    'g',
    'h',    'i',    'j',    'k',    'l',    'm',    'n',    'o',
    'p',    'q',    'r',    's',    't',    'u',    'v',    'w',
    'x',    'y',    'z',    '{',    '|',    '~',    '}',    DEL);
```

Note that type character is an example of an enumeration type containing a mixture of identifiers and characters. Also, the characters '0' and '1' are members of both bit and character. Where '0' or '1' occur in a program, the context will be used to determine which type is being used.

2.2.5. Arrays

An array in VHDL is an indexed collection of elements all of the same type. Arrays may be one-dimensional (with one index) or multi-dimensional (with a number of indices). In addition, an array type may be constrained, in which the bounds for an index are established when the type is defined, or unconstrained, in which the bounds are established subsequently.

The syntax for declaring an array type is:

```
array_type_definition ::=
    unconstrained_array_definition | constrained_array_definition
unconstrained_array_definition ::=
    array ( index_subtype_definition { , index_subtype_definition } )
    of element_subtype_indication
constrained_array_definition ::=
    array index_constraint of element_subtype_indication
index_subtype_definition ::= type_mark range <>
index_constraint ::= ( discrete_range { , discrete_range } )
discrete_range ::= discrete_subtype_indication | range
```


Subtypes, referred to in this syntax specification, will be discussed in detail in Section 2.2.7.

Some examples of constrained array type declarations:

```
type word is array (31 downto 0) of bit;
type memory is array (address) of word;
type transform is array (1 to 4, 1 to 4) of real;
type register_bank is array (byte range 0 to 132) of integer;
```

An example of an unconstrained array type declaration:

```
type vector is array (integer range <>) of real;
```

The symbol '<>' (called a box) can be thought of as a place-holder for the index range, which will be filled in later when the array type is used. For example, an object might be declared to be a vector of 20 elements by giving its type as:

```
vector(1 to 20)
```

There are two predefined array types, both of which are unconstrained. They are defined as:

```
type string is array (positive range <>) of character;
type bit_vector is array (natural range <>) of bit;
```

The types positive and natural are subtypes of integer, defined in Section 2.2.7 below. The type bit_vector is particularly useful in modeling binary coded representations of values in simulations of digital systems.

An element of an array object can be referred to by indexing the name of the object. For example, suppose a and b are one- and two-dimensional array objects respectively. Then the indexed names a(1) and b(1, 1) refer to elements of these arrays. Furthermore, a contiguous slice of a one-dimensional array can be referred to by using a range as an index. For example a(8 to 15) is an eight-element array which is part of the array a.

Sometimes you may need to write a literal value of an array type. This can be done using an array aggregate, which is a list of element values. Suppose we have an array type declared as:

```
type a is array (1 to 4) of character;
```

and we want to write a value of this type containing the elements 'f', 'o', 'o', 'd' in that order. We could write an aggregate with *positional* association as follows:

```
('f', 'o', 'o', 'd')
```

in which the elements are listed in the order of the index range, starting with the left bound of the range. Alternatively, we could write an aggregate with *named* association:

```
(1 => 'f', 3 => 'o', 4 => 'd', 2 => 'o')
```

In this case, the index for each element is explicitly given, so the elements can be in any order. Positional and named association can be mixed within an aggregate, provided all the positional associations come first. Also, the word **others** can be used in place of an index in a named association, indicating a value to be used for all elements not explicitly mentioned. For example, the same value as above could be written as:

```
('f', 4 => 'd', others => 'o')
```

2.2.6. Records

VHDL provides basic facilities for records, which are collections of named elements of possibly different types. The syntax for declaring record types is:

```
record_type_definition ::=
    record
        element_declaration
        { element_declaration }
    end record

element_declaration ::= identifier_list : element_subtype_definition ;
identifier_list ::= identifier { , identifier }
element_subtype_definition ::= subtype_indication
```

An example record type declaration:

```
type instruction is
    record
        op_code : processor_op;
        address_mode : mode;
        operand1, operand2: integer range 0 to 15;
    end record;
```

When you need to refer to a field of a record object, you use a selected name. For example, suppose that *r* is a record object containing a field called *f*. Then the name *r.f* refers to that field.

As for arrays, aggregates can be used to write literal values for records. Both positional and named association can be used, and the same rules apply, with record field names being used in place of array index names.

2.2.7. Subtypes

The use of a subtype allows the values taken on by an object to be restricted or constrained subset of some base type. The syntax for declaring a subtype is:

```
subtype_declaration ::= subtype identifier is subtype_indication ;
subtype_indication ::= [ resolution_function_name ] type_mark [ constraint ]
type_mark ::= type_name | subtype_name
constraint ::= range_constraint | index_constraint
```

There are two cases of subtypes. Firstly a subtype may constrain values from a scalar type to be within a specified range (a range constraint). For example:

```
subtype pin_count is integer range 0 to 400;
subtype digits is character range '0' to '9';
```

Secondly, a subtype may constrain an otherwise unconstrained array type by specifying bounds for the indices. For example:

```
subtype id is string(1 to 20);
subtype word is bit_vector(31 downto 0);
```

There are two predefined numeric subtypes, defined as:

```
subtype natural is integer range 0 to highest_integer
subtype positive is integer range 1 to highest_integer
```

2.2.8. Object Declarations

An object is a named item in a VHDL description which has a value of a specified type. There are three classes of objects: constants, variables and signals. Only the first two will be discussed in this section; signals will be covered in Section 3.2.1. Declaration and use of constants and variables is very much like their use in programming languages.

A constant is an object which is initialised to a specified value when it is created, and which may not be subsequently modified. The syntax of a constant declaration is:

```
constant_declaration ::=
    constant identifier_list : subtype_indication [ := expression ] ;
```

Constant declarations with the initialising expression missing are called deferred constants, and may only appear in package declarations (see Section 2.5.3). The initial value must be given in the corresponding package body. Some examples:

```
constant e : real := 2.71828;
constant delay : Time := 5 ns;
constant max_size : natural;
```

A variable is an object whose value may be changed after it is created. The syntax for declaring variables is:

```
variable_declaration ::=
    variable identifier_list : subtype_indication [ := expression ] ;
```

The initial value expression, if present, is evaluated and assigned to the variable when it is created. If the expression is absent, a default value is assigned when the variable is created. The default value for scalar types is the leftmost value for the type, that is the first in the list of an enumeration type, the lowest in an ascending range, or the highest in a descending range. If the variable is a composite type, the default value is the composition of the default values for each element, based on the element types.

Some examples of variable declarations:

```
variable count : natural := 0;
variable trace : trace_array;
```

Assuming the type `trace_array` is an array of boolean, then the initial value of the variable `trace` is an array with all elements having the value false.

Given an existing object, it is possible to give an alternate name to the object or part of it. This is done using an alias declaration. The syntax is:

```
alias_declaration ::= alias identifier : subtype_indication is name ;
```

A reference to an alias is interpreted as a reference to the object or part corresponding to the alias. For example:

```
variable instr : bit_vector(31 downto 0);
alias op_code : bit_vector(7 downto 0) is instr(31 downto 24);
```

declares the name `op_code` to be an alias for the left-most eight bits of `instr`.

2.2.9. Attributes

Types and objects declared in a VHDL description can have additional information, called attributes, associated with them. There are a number of standard pre-defined attributes, and some of those for types and arrays

are discussed here. An attribute is referenced using the "" notation. For example,

thing'attr

refers to the attribute attr of the type or object thing.

Firstly, for any scalar type or subtype T, the following attributes can be used:

<u>Attribute</u>	<u>Result</u>
T'left	Left bound of T
T'right	Right bound of T
T'low	Lower bound of T
T'high	Upper bound of T

For an ascending range, T'left = T'low, and T'right = T'high. For a descending range, T'left = T'high, and T'right = T'low.

Secondly, for any discrete or physical type or subtype T, X a member of T, and N an integer, the following attributes can be used:

<u>Attribute</u>	<u>Result</u>
T'pos(X)	Position number of X in T
T'val(N)	Value at position N in T
T'leftof(X)	Value in T which is one position left from X
T'rightof(X)	Value in T which is one position right from X
T'pred(X)	Value in T which is one position lower than X
T'succ(X)	Value in T which is one position higher than X

For an ascending range, T'leftof(X) = T'pred(X), and T'rightof(X) = T'succ(X). For a descending range, T'leftof(X) = T'succ(X), and T'rightof(X) = T'pred(X).

Thirdly, for any array type or object A, and N an integer between 1 and the number of dimensions of A, the following attributes can be used:

<u>Attribute</u>	<u>Result</u>
A'left(N)	Left bound of index range of dim'n N of A
A'right(N)	Right bound of index range of dim'n N of A
A'low(N)	Lower bound of index range of dim'n N of A
A'high(N)	Upper bound of index range of dim'n N of A
A'range(N)	Index range of dim'n N of A
A'reverse_range(N)	Reverse of index range of dim'n N of A
A'length(N)	Length of index range of dim'n N of A

2.3. Expressions and Operators

Expressions in VHDL are much like expressions in other programming languages. An expression is a formula combining primaries with operators. Primaries include names of objects, literals, function calls and parenthesized expressions. Operators are listed in Table 2-1 in order of decreasing precedence.

The logical operators **and**, **or**, **nand**, **nor**, **xor** and **not** operate on values of type bit or boolean, and also on one-dimensional arrays of these types. For array operands, the operation is applied between corresponding elements of each array, yielding an array of the same length as the result. For bit and

Highest precedence:	**	abs	not				
	*	/	mod	rem			
	+ (sign)	- (sign)					
	+	-	&				
	=	/=	<	<=	>	>=	
Lowest precedence:	and	or	nand	nor	xor		

Table 7-1. Operators and precedence.

boolean operands, **and**, **or**, **nand**, and **nor** are ‘short-circuit’ operators, that is they only evaluate their right operand if the left operand does not determine the result. So **and** and **nand** only evaluate the right operand if the left operand is true or '1', and **or** and **nor** only evaluate the right operand if the left operand is false or '0'.

The relational operators **=**, **/=**, **<**, **<=**, **>** and **>=** must have both operands of the same type, and yield boolean results. The equality operators (**=** and **/=**) can have operands of any type. For composite types, two values are equal if all of their corresponding elements are equal. The remaining operators must have operands which are scalar types or one-dimensional arrays of discrete types.

The sign operators (**+** and **-**) and the addition (**+**) and subtraction (**-**) operators have their usual meaning on numeric operands. The concatenation operator (**&**) operates on one-dimensional arrays to form a new array with the contents of the right operand following the contents of the left operand. It can also concatenate a single new element to an array, or two individual elements to form an array. The concatenation operator is most commonly used with strings.

The multiplication (*****) and division (**/**) operators work on integer, floating point and physical types. The modulus (**mod**) and remainder (**rem**) operators only work on integer types. The absolute value (**abs**) operator works on any numeric type. Finally, the exponentiation (******) operator can have an integer or floating point left operand, but must have an integer right operand. A negative right operand is only allowed if the left operand is a floating point number.

2.4. Sequential Statements

VHDL contains a number of facilities for modifying the state of objects and controlling the flow of execution of models. These are discussed in this section.

2.4.1. Variable Assignment

As in other programming languages, a variable is given a new value using an assignment statement. The syntax is:

```
variable_assignment_statement ::= target := expression ;
target ::= name | aggregate
```

In the simplest case, the target of the assignment is an object name, and the value of the expression is given to the named object. The object and the value must have the same base type.

If the target of the assignment is an aggregate, then the elements listed must be object names, and the value of the expression must be a composite value of the same type as the aggregate. Firstly, all the names in the aggregate are evaluated, then the expression is evaluated, and lastly the components of the expression value are assigned to the named variables. This is effectively a parallel assignment. For example, if a variable *r* is a record with two fields *a* and *b*, then they could be exchanged by writing

```
(a => r.b, b => r.a) := r
```

(Note that this is an example to illustrate how such an assignment works; it is not an example of good programming practice!)

2.4.2. If Statement

The if statement allows selection of statements to execute depending on one or more conditions. The syntax is:

```
if_statement ::=
    if condition then
        sequence_of_statements
    { elsif condition then
        sequence_of_statements }
    [ else
        sequence_of_statements ]
    end if ;
```

The conditions are expressions resulting in boolean values. The conditions are evaluated successively until one found that yields the value true. In that case the corresponding statement list is executed. Otherwise, if the else clause is present, its statement list is executed.

2.4.3. Case Statement

The case statement allows selection of statements to execute depending on the value of a selection expression. The syntax is:

```
case_statement ::=
    case expression is
        case_statement_alternative
        { case_statement_alternative }
    end case ;
case_statement_alternative ::=
    when choices =>
        sequence_of_statements
choices ::= choice { | choice }
choice ::=
    simple_expression
    | discrete_range
    / element_simple_name
    | others
```

The selection expression must result in either a discrete type, or a one-dimensional array of characters. The alternative whose choice list includes the value of the expression is selected and the statement list executed. Note that all the choices must be distinct, that is, no value may be duplicated. Furthermore, all values must be represented in the choice lists, or the special choice **others** must be included as the last alternative. If no choice list includes the value of the expression, the others alternative is selected. If the expression results in an array, then the choices may be strings or bit strings.

Some examples of case statements:

```

case element_colour of
  when red =>
    statements for red;
  when green | blue =>
    statements for green or blue;
  when orange to turquoise =>
    statements for these colours;
end case;

case opcode of
  when X"00" => perform_add;
  when X"01" => perform_subtract;
  when others => signal_illegal_opcode;
end case;

```

2.4.4. Loop Statements

VHDL has a basic loop statement, which can be augmented to form the usual while and for loops seen in other programming languages. The syntax of the loop statement is:

```

loop_statement ::=
  [ loop_label : ]
  [ iteration_scheme ] loop
    sequence_of_statements
  end loop [ loop_label ] ;

iteration_scheme ::=
  while condition
  | for loop_parameter_specification

parameter_specification ::=
  identifier in discrete_range

```

If the iteration scheme is omitted, we get a loop which will repeat the enclosed statements indefinitely. An example of such a basic loop is:

```

loop
  do_something;
end loop;

```

The while iteration scheme allows a test condition to be evaluated before each iteration. The iteration only proceeds if the test evaluates to true. If the test is false, the loop statement terminates. An example:

```

while index < length and str(index) /= ' ' loop
  index := index + 1;
end loop;

```

The for iteration scheme allows a specified number of iterations. The loop parameter specification declares an object which takes on successive values from the given range for each iteration of the loop. Within the statements enclosed in the loop, the object is treated as a constant, and so may not be assigned to. The object does not exist beyond execution of the loop statement. An example:

```

for item in 1 to last_item loop
  table(item) := 0;
end loop;

```

There are two additional statements which can be used inside a loop to modify the basic pattern of iteration. The 'next' statement terminates execution of the current iteration and starts the subsequent iteration. The

'exit' statement terminates execution of the current iteration and terminates the loop. The syntax of these statements is:

```
next_statement ::= next [ loop_label ] [ when condition ] ;
exit_statement ::= exit [ loop_label ] [ when condition ] ;
```

If the loop label is omitted, the statement applies to the inner-most enclosing loop, otherwise it applies to the named loop. If the when clause is present but the condition is false, the iteration continues normally. Some examples:

```
for i in 1 to max_str_len loop
    a(i) := buf(i);
    exit when buf(i) = NUL;
end loop;

outer_loop : loop
    inner_loop : loop
        do_something;
        next outer_loop when temp = 0;
        do_something_else;
    end loop inner_loop;
end loop outer_loop;
```

2.4.5. Null Statement

The null statement has no effect. It may be used to explicitly show that no action is required in certain cases. It is most often used in case statements, where all possible values of the selection expression must be listed as choices, but for some choices no action is required. For example:

```
case controller_command is
    when forward => engage_motor_forward;
    when reverse => engage_motor_reverse;
    when idle => null;
end case;
```

2.4.6. Assertions

An assertion statement is used to verify a specified condition and to report if the condition is violated. The syntax is:

```
assertion_statement ::=
    assert condition
        [ report expression ]
        [ severity expression ] ;
```

If the report clause is present, the result of the expression must be a string. This is a message which will be reported if the condition is false. If it is omitted, the default message is "Assertion violation". If the severity clause is present the expression must be of the type severity_level. If it is omitted, the default is error. A simulator may terminate execution if an assertion violation occurs and the severity value is greater than some implementation dependent threshold. Usually the threshold will be under user control.

2.5. Subprograms and Packages

Like other programming languages, VHDL provides subprogram facilities in the form of procedures and functions. VHDL also provided a package facility for collecting declarations and objects into modular units. Packages also provide a measure of data abstraction and information hiding.

2.5.1. Procedures and Functions

Procedure and function subprograms are declared using the syntax:

```
subprogram_declaration ::= subprogram_specification ;
subprogram_specification ::=
    procedure designator [ ( formal_parameter_list ) ]
    | function designator [ ( formal_parameter_list ) ] return type_mark
```

A subprogram declaration in this form simply names the subprogram and specifies the parameters required. The body of statements defining the behaviour of the subprogram is deferred. For function subprograms, the declaration also specifies the type of the result returned when the function is called. This form of subprogram declaration is typically used in package specifications (see Section 2.5.3), where the subprogram body is given in the package body, or to define mutually recursive procedures.

The syntax for specifying the formal parameters of a subprogram is:

```
formal_parameter_list ::= parameter_interface_list
interface_list ::= interface_element { ; interface_element }
interface_element ::= interface_declaration
interface_declaration ::=
    interface_constant_declaration
    | interface_signal_declaration
    | interface_variable_declaration
interface_constant_declaration ::=
    [ constant ] identifier_list : [ in ] subtype_indication [ := static_expression ]
interface_variable_declaration ::=
    [ variable ] identifier_list : [ mode ] subtype_indication [ := static_expression ]
```

For now we will only consider constant and variable parameters, although signals can also be used (see Chapter 3). Some examples will clarify this syntax. Firstly, a simple example of a procedure with no parameters:

```
procedure reset;
```

This simply defines `reset` as a procedure with no parameters, whose statement body will be given subsequently in the VHDL program. A procedure call to `reset` would be:

```
reset;
```

Secondly, here is a declaration of a procedure with some parameters:

```
procedure increment_reg(variable reg : inout word_32;
                       constant incr : in integer := 1);
```

In this example, the procedure `increment_reg` has two parameters, the first called `reg` and the second called `incr`. `Reg` is a variable parameter, which means that in the subprogram body, it is treated as a variable object and may be assigned to. This means that when the procedure is called, the actual parameter associated with `reg` must itself be a variable. The mode of `reg` is `inout`, which means that `reg` can be both read and assigned to. Other possible modes for subprogram parameters are `in`, which means that the parameter may only be read, and `out`, which means that the parameter may only be assigned to. If the mode is `inout` or `out`, then the word **variable** can be omitted and is assumed.

The second parameter, `incr`, is a constant parameter, which means that it is treated as a constant object in the subprogram statement body, and may not be assigned to. The actual parameter associated with `incr` when the procedure is called must be an expression. Given the mode of the

parameter, **in**, the word **constant** could be omitted and assumed. The expression after the assignment operator is a default expression, which is used if no actual parameter is associated with `incr` in a call to the procedure.

A call to a subprogram includes a list of actual parameters to be associated with the formal parameters. This association list can be position, named, or a combination of both. (Compare this with the format of aggregates for values of composite types.) A call with positional association lists the actual parameters in the same order as the formals. For example:

```
increment_reg(index_reg, offset-2);    -- add value to index_reg
increment_reg(prog_counter);           -- add 1 (default) to prog_counter
```

A call with named association explicitly gives the formal parameter name to be associated with each actual parameter, so the parameters can be in any order. For example:

```
increment_reg(incr => offset-2, reg => index_reg);
increment_reg(reg => prog_counter);
```

Note that the second call in each example does not give a value for the formal parameter `incr`, so the default value is used.

Thirdly, here is an example of function subprogram declaration:

```
function byte_to_int(byte : word_8) return integer;
```

The function has one parameter. For functions, the parameter mode must be **in**, and this is assumed if not explicitly specified. If the parameter class is not specified it is assumed to be **constant**. The value returned by the body of this function must be an integer.

When the body of a subprogram is specified, the syntax used is:

```
subprogram_body ::=
    subprogram_specification is
        subprogram_declarative_part
    begin
        subprogram_statement_part
    end [ designator ] ;

subprogram_declarative_part ::= { subprogram_declarative_item }
subprogram_statement_part ::= { sequential_statement }
subprogram_declarative_item ::=
    subprogram_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | alias_declaration
```

The declarative items listed after the subprogram specification declare things which are to be used locally within the subprogram body. The names of these items are not visible outside of the subprogram, but are visible inside locally declared subprograms. Furthermore, these items shadow any things with the same names declared outside the subprogram.

When the subprogram is called, the statements in the body are executed until either the end of the statement list is encountered, or a return statement is executed. The syntax of a return statement is:

```
return_statement ::= return [ expression ] ;
```

If a return statement occurs in a procedure body, it must not include an expression. There must be at least one return statement in a function body, it must have an expression, and the function must complete by executing a return statement. The value of the expression is the value returned to the function call.

Another point to note about function subprograms is that they may not have any side-effects. This means that no visible variable declared outside the function body may be assigned to or altered by the function. This includes passing a non-local variable to a procedure as a variable parameter with mode **out** or **inout**. The important result of this rule is that functions can be called without them having any effect on the environment of the call.

An example of a function body:

```
function byte_to_int(byte : word_8) return integer is
  variable result : integer := 0;
begin
  for index in 0 to 7 loop
    result := result*2 + bit'pos(byte(index));
  end loop;
  return result;
end byte_to_int;
```

2.5.2. Overloading

VHDL allows two subprograms to have the same name, provided the number or base types of parameters differs. The subprogram name is then said to be overloaded. When a subprogram call is made using an overloaded name, the number of actual parameters, their order, their base types and the corresponding formal parameter names (if named association is used) are used to determine which subprogram is meant. If the call is a function call, the result type is also used. For example, suppose we declared the two subprograms:

```
function check_limit(value : integer) return boolean;
function check_limit(value : word_32) return boolean;
```

Then which of the two functions is called depends on whether a value of type integer or word_8 is used as the actual parameter. So

```
test := check_limit(4095)
```

would call the first function, and

```
test := check_limit(X"0000_0FFF")
```

would call the second function.

The designator used to define a subprogram can be either an identifier or a string representing any of the operator symbols listed in Section 2.3. The latter case allows extra operand types to be defined for those operators. For example, the addition operator might be overloaded to add word_32 operands by declaring a function:

```
function "+" (a, b : word_32) return word_32 is
begin
  return int_to_word_32( word_32_to_int(a) + word_32_to_int(b) );
end "+";
```

Within the body of this function, the addition operator is used to add integers, since its operands are both integers. However, in the expression:

```
X"1000_0010" + X"0000_FFD0"
```

the newly declared function is called, since the operands to the addition operator are both of type `word_32`. Note that it is also possible to call operators using the prefix notation used for ordinary subprogram calls, for example:

```
"+" (X"1000_0010", X"0000_FFD0")
```

2.5.3. Package and Package Body Declarations

A package is a collection of types, constants, subprograms and possibly other things, usually intended to implement some particular service or to isolate a group of related items. In particular, the details of constant values and subprogram bodies can be hidden from users of a package, with only their interfaces made visible.

A package may be split into two parts: a package declaration, which defines its interface, and a package body, which defines the deferred details. The body part may be omitted if there are no deferred details. The syntax of a package declaration is:

```
package_declaration ::=
    package identifier is
        package_declarative_part
    end [ package_simple_name ] ;
package_declarative_part ::= { package_declarative_item }
package_declarative_item ::=
    subprogram_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | alias_declaration
    | use_clause
```

The declarations define things which are to be visible to users of the package, and which are also visible inside the package body. (There are also other kinds of declarations which can be included, but they are not discussed here.)

An example of a package declaration:

```
package data_types is
    subtype address is bit_vector(24 downto 0);
    subtype data is bit_vector(15 downto 0);
    constant vector_table_loc : address;
    function data_to_int(value : data) return integer;
    function int_to_data(value : integer) return data;
end data_types;
```

In this example, the value of the constant `vector_table_loc` and the bodies of the two functions are deferred, so a package body needs to be given.

The syntax for a package body is:

```
package_body ::=
    package body package_simple_name is
        package_body_declarative_part
    end [ package_simple_name ] ;
package_body_declarative_part ::= { package_body_declarative_item }
```

```

package_body_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | alias_declaration
    | use_clause

```

Note that subprogram bodies may be included in a package body, whereas only subprogram interface declarations may be included in the package interface declaration.

The body for the package `data_types` shown above might be written as:

```

package body data_types is
    constant vector_table_loc : address := X"FFFF00";
    function data_to_int(value : data) return integer is
        body of data_to_int
    end data_to_int;
    function int_to_data(value : integer) return data is
        body of int_to_data
    end int_to_data;
end data_types;

```

In this package body, the value for the constant is specified, and the function bodies are given. The subtype declarations are not repeated, as those in the package declarations are visible in the package body.

2.5.4. Package Use and Name Visibility

Once a package has been declared, items declared within it can be used by prefixing their names with the package name. For example, given the package declaration in Section 2.4.3 above, the items declared might be used as follows:

```

variable PC : data_types.address;
int_vector_loc := data_types.vector_table_loc + 4*int_level;
offset := data_types.data_to_int(offset_reg);

```

Often it is convenient to be able to refer to names from a package without having to qualify each use with the package name. This may be done using a use clause in a declaration region. The syntax is:

```

use_clause ::= use selected_name { , selected_name } ;
selected_name ::= prefix . suffix

```

The effect of the use clause is that all of the listed names can subsequently be used without having to prefix them. If all of the declared names in a package are to be used in this way, you can use the special suffix `all`, for example:

```

use data_types.all;

```

3. VHDL Describes Structure

In Section 1.1 we introduced some terminology for describing the structure of a digital system. In this chapter, we will look at how structure is described in VHDL.

3.1. Entity Declarations

A digital system is usually designed as a hierarchical collection of modules. Each module has a set of ports which constitute its interface to the outside world. In VHDL, an *entity* is such a module which may be used as a component in a design, or which may be the top level module of the design.

The syntax for declaring an entity is:

```
entity_declaration ::=
    entity identifier is
        entity_header
        entity_declarative_part
    [ begin
        entity_statement_part ]
    end [ entity_simple_name ] ;

entity_header ::=
    [ formal_generic_clause ]
    [ formal_port_clause ]

generic_clause ::= generic ( generic_list ) ;
generic_list ::= generic_interface_list
port_clause ::= port ( port_list ) ;
port_list ::= port_interface_list
entity_declarative_part ::= { entity_declarative_item }
```

The entity declarative part may be used to declare items which are to be used in the implementation of the entity. Usually such declarations will be included in the implementation itself, so they are only mentioned here for completeness. Also, the optional statements in the entity declaration may be used to define some special behaviour for monitoring operation of the entity. Discussion of these will be deferred until Section 6.5.

The entity header is the most important part of the entity declaration. It may include specification of *generic constants*, which can be used to control the structure and behaviour of the entity, and *ports*, which channel information into and out of the entity.

The generic constants are specified using an interface list similar to that of a subprogram declaration. All of the items must be of class constant. As a reminder, the syntax of an interface constant declaration is:

```
interface_constant_declaration ::=
    [ constant ] identifier_list : [ in ] subtype_indication [ := static_expression ]
```

The actual value for each generic constant is passed in when the entity is used as a component in a design.

The entity ports are also specified using an interface list, but the items in the list must all be of class **signal**. This is a new kind of interface item not previously discussed. The syntax is:

```
interface_signal_declaration ::=
  [ signal ] identifier_list : [ mode ] subtype_indication [ bus ]
  [ := static_expression ]
```

Since the class must be **signal**, the word **signal** can be omitted and is assumed. The word **bus** may be used if the port is to be connected to more than one output (see Sections 6.1 and 6.2). As with generic constants the actual signals to be connected to the ports are specified when the entity is used as a component in a design.

To clarify this discussion, here are some examples of entity declarations:

```
entity processor is
  generic (max_clock_freq : frequency := 30 MHz);
  port (clock : in bit;
        address : out integer;
        data : inout word_32;
        control : out proc_control;
        ready : in bit);
end processor;
```

In this case, the generic constant `max_clock_freq` is used to specify the timing behaviour of the entity. The code describing the entity's behaviour would use this value to determine delays in changing signal values.

Next, an example showing how generic parameters can be used to specify a class of entities with varying structure:

```
entity ROM is
  generic (width, depth : positive);
  port (enable : in bit;
        address : in bit_vector(depth-1 downto 0);
        data : out bit_vector(width-1 downto 0));
end ROM;
```

Here, the two generic constants are used to specify the number of data bits and address bits respectively for the read-only memory. Note that no default value is given for either of these constants. This means that when the entity is used as a component, actual values must be supplied for them.

Finally an example of an entity declaration with no generic constants or

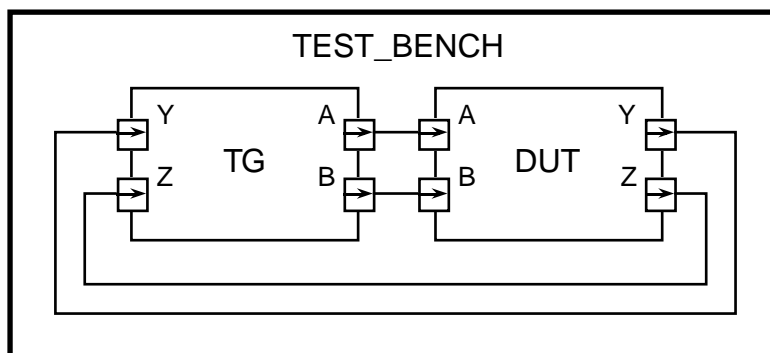


Figure 3-1. Test bench circuit.

ports:

```
entity test_bench is
end test_bench;
```

Though this might at first seem to be a pointless example, in fact it illustrates a common use of entities, shown in Figure3-1. A top-level entity for a design under test (DUT) is used as a component in a test bench circuit with another entity (TG) whose purpose is to generate test values. The values on signals can be traced using a simulation monitor, or checked directly by the test generator. No external connections from the test bench are needed, hence it has no ports.

3.2. Architecture Declarations

Once an entity has had its interface specified in an entity declaration, one or more implementations of the entity can be described in *architecture* bodies. Each architecture body can describe a different view of the entity. For example, one architecture body may purely describe the behaviour using the facilities covered in Chapters 2 and 4, whereas others may describe the structure of the entity as a hierarchically composed collection of components. In this section, we will only cover structural descriptions, deferring behaviour descriptions until Chapter4.

An architecture body is declared using the syntax:

```
architecture_body ::=
    architecture identifier of entity_name is
        architecture_declarative_part
    begin
        architecture_statement_part
    end [ architecture_simple_name ] ;
architecture_declarative_part ::= { block_declarative_item }
architecture_statement_part ::= { concurrent_statement }
block_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | alias_declaration
    | component_declaration
    | configuration_specification
    | use_clause
concurrent_statement ::=
    block_statement
    | component_instantiation_statement
```

The declarations in the architecture body define items that will be used to construct the design description. In particular, signals and components may be declared here and used to construct a structural description in terms of component instances, as illustrated in Section1.4. These are discussed in more detail in the next sections.

3.2.1. Signal Declarations

Signals are used to connect submodules in a design. They are declared using the syntax:


```

signal_declaration ::=
    signal identifier_list : subtype_indication [ signal_kind ] [ := expression ] ;
signal_kind ::= register | bus

```

Use of the signal kind specification is covered in Section 6.2. Omitting the signal kind results in an ordinary signal of the subtype specified. The expression in the declaration is used to give the signal an initial value during the initialization phase of simulation. If the expression is omitted, a default initial value will be assigned.

One important point to note is that ports of an object are treated exactly as signals within that object.

3.2.2. Blocks

The submodules in an architecture body can be described as blocks. A block is a unit of module structure, with its own interface, connected to other blocks or ports by signals. A block is specified using the syntax:

```

block_statement ::=
    block_label :
        block [ ( guard_expression ) ]
            block_header
            block_declarative_part
        begin
            block_statement_part
        end block [ block_label ] ;

block_header ::=
    [ generic_clause
    [ generic_map_aspect ; ] ]
    [ port_clause
    [ port_map_aspect ; ] ]

generic_map_aspect ::= generic map ( generic_association_list )
port_map_aspect ::= port map ( port_association_list )
block_declarative_part ::= { block_declarative_item }
block_statement_part ::= { concurrent_statement }

```

The guard expression is not covered in this booklet, and may be omitted. The block header defines the interface to the block in much the same way as an entity header defines the interface to an entity. The generic association list specifies values for the generic constants, evaluated in the context of the enclosing block or architecture body. The port map association list specifies which actual signals or ports from the enclosing block or architecture body are connected to the block's ports. Note that a block statement part may also contain block statements, so a design can be composed of a hierarchy of blocks, with behavioural descriptions at the bottom level of the hierarchy.

As an example, suppose we want to describe a structural architecture of the processor entity example in Section 3.1. If we separate the processor into a control unit and a data path section, we can write a description as a pair of interconnected blocks, as shown in Figure 3-2.

The control unit block has ports `clk`, `bus_control` and `bus_ready`, which are connected to the processor entity ports. It also has an output port for controlling the data path, which is connected to a signal declared in the architecture. That signal is also connected to a control port on the data path block. The address and data ports of the data path block are connected to the corresponding entity ports. The advantage of this modular decomposition is that each of the blocks can then be developed

```

architecture block_structure of processor is
    type data_path_control is ... ;
    signal internal_control : data_path_control;
begin
    control_unit : block
        port (clk : in bit;
            bus_control : out proc_control;
            bus_ready : in bit;
            control : out data_path_control);
        port map (clk => clock,
            bus_control => control, bus_ready => ready;
            control => internal_control);
        declarations for control_unit
    begin
        statements for control_unit
    end block control_unit;
    data_path : block
        port (address : out integer;
            data : inout word_32;
            control : in data_path_control);
        port map (address => address, data => data,
            control => internal_control);
        declarations for data_path
    begin
        statements for data_path
    end block data_path;
end block_structure;

```

Figure3-2. Structural architecture of processor example.

independently, with the only effects on other blocks being well defined through their interfaces.

3.2.3. Component Declarations

An architecture body can also make use of other entities described separately and placed in design libraries. In order to do this, the architecture must declare a component, which can be thought of as a template defining a virtual design entity, to be instantiated within the architecture. Later, a configuration specification (see Section3.3) can be used to specify a matching library entity to use. The syntax of a component declaration is:

```

component_declaration ::=
    component identifier
        [ local_generic_clause ]
        [ local_port_clause ]
    end component ;

```

Some examples of component declarations:

```

component nand3
    generic (Tpd : Time := 1 ns);
    port (a, b, c : in logic_level;
        y : out logic_level);
end component;

```

```

component read_only_memory
  generic (data_bits, addr_bits : positive);
  port (en : in bit;
        addr : in bit_vector(depth-1 downto 0);
        data : out bit_vector(width-1 downto 0) );
end component;

```

The first example declares a three-input gate with a generic parameter specifying its propagation delay. Different instances can later be used with possibly different propagation delays. The second example declares a read-only memory component with address depth and data width dependent on generic constants. This component could act as a template for the ROM entity described in Section 3.1.

3.2.4. Component Instantiation

A component defined in an architecture may be instantiated using the syntax:

```

component_instantiation_statement ::=
  instantiation_label :
    component_name
    [ generic_map_aspect ]
    [ port_map_aspect ];

```

This indicates that the architecture contains an instance of the named component, with actual values specified for generic constants, and with the component ports connected to actual signals or entity ports.

The example components declared in the previous section might be instantiated as:

```

enable_gate: nand3
  port map (a => en1, b => en2, c => int_req, y => interrupt);

parameter_rom: read_only_memory
  generic map (data_bits => 16, addr_bits => 8);
  port map (en => rom_sel, data => param, addr => a(7 downto 0));

```

In the first instance, no generic map specification is given, so the default value for the generic constant *Tpd* is used. In the second instance, values are specified for the address and data port sizes. Note that the actual signal associated with the port *addr* is a slice of an array signal. This illustrates that a port which is an array can be connected to part of a signal which is a larger array, a very common practice with bus signals.

4. VHDL Describes Behaviour

In Section 1.2 we stated that the behaviour of a digital system could be described in terms of programming language notation. The familiar sequential programming language aspects of VHDL were covered in detail in Chapter 2. In this chapter, we describe how these are extended to include statements for modifying values on signals, and means of responding to the changing signal values.

4.1. Signal Assignment

A signal assignment schedules one or more transactions to a signal (or port). The syntax of a signal assignment is:

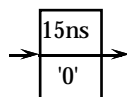
```
signal_assignment_statement ::= target <= [ transport ] waveform ;  
target ::= name | aggregate  
waveform ::= waveform_element { , waveform_element }  
waveform_element ::=  
    value_expression [ after time_expression ]  
    | null [ after time_expression ]
```

The target must represent a signal, or be an aggregate of signals (see also variable assignments, Section 2.4.1). If the time expression for the delay is omitted, it defaults to 0 fs. This means that the transaction will be scheduled for the same time as the assignment is executed, but during the next simulation cycle.

Each signal has associated with it a *projected output waveform*, which is a list of transactions giving future values for the signal. A signal assignment adds transactions to this waveform. So, for example, the signal assignment:

```
s <= '0' after 10 ns;
```

will cause the signal enable to assume the value true 10 ns after the assignment is executed. We can represent the projected output waveform graphically by showing the transactions along a time axis. So if the above assignment were executed at time 5 ns, the projected waveform would be:

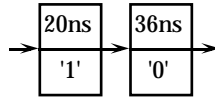


When simulation time reaches 15 ns, this transaction will be processed and the signal updated.

Suppose then at time 16 ns, the assignment:

```
s <= '1' after 4 ns, '0' after 20 ns;
```

were executed. The two new transactions are added to the projected output waveform:

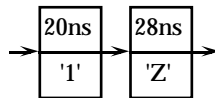


Note that when multiple transactions are listed in a signal assignment, the delay times specified must be in ascending order.

If a signal assignment is executed, and there are already old transactions from a previous assignment on the projected output waveform, then some of the old transactions may be deleted. The way this is done depends on whether the word **transport** is included in the new assignment. If it is included, the assignment is said to use *transport delay*. In this case, all old transactions scheduled to occur after the first new transaction are deleted before the new transactions are added. It is as though the new transactions supercede the old ones. So given the projected output waveform shown immediately above, if the assignment:

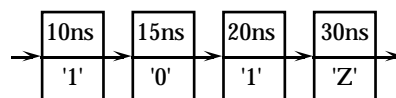
```
s <= transport 'Z' after 10 ns;
```

were executed at time 18 ns, then the transaction scheduled for 36 ns would be deleted, and the projected output waveform would become:



The second kind of delay, *inertial delay*, is used to model devices which do not respond to input pulses shorter than their output delay. An inertial delay is specified by omitting the word **transport** from the signal assignment. When an inertial delay transaction is added to a projected output waveform, firstly all old transactions scheduled to occur after the new transaction are deleted, and the new transaction is added, as in the case of transport delay. Next, all old transactions scheduled to occur before the new transaction are examined. If there are any with a different value from the new transaction, then all transactions up to the last one with a different value are deleted. The remaining transactions with the same value are left.

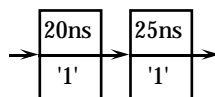
To illustrate this, suppose the projected output waveform at time 0 ns is:



and the assignment:

```
s <= '1' after 25 ns;
```

is executed also at 0 ns. Then the new projected output waveform is:



When a signal assignment with multiple waveform elements is specified with inertial delay, only the first transaction uses inertial delay; the rest are treated as being transport delay transactions.

4.2. Processes and the Wait Statement

The primary unit of behavioural description in VHDL is the *process*. A process is a sequential body of code which can be activated in response to changes in state. When more than one process is activated at the same

time, they execute concurrently. A process is specified in a process statement, with the syntax:

```

process_statement ::=
    [ process_label : ]
    process [ ( sensitivity_list ) ]
    process_declarative_part
    begin
    process_statement_part
    end process [ process_label ] ;
process_declarative_part ::= { process_declarative_item }
process_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | alias_declaration
    | use_clause
process_statement_part ::= { sequential_statement }
sequential_statement ::=
    wait_statement
    | assertion_statement
    | signal_assignment_statement
    | variable_assignment_statement
    | procedure_call_statement
    | if_statement
    | case_statement
    | loop_statement
    | next_statement
    | exit_statement
    | return_statement
    | null_statement

```

A process statement is a concurrent statement which can be used in an architecture body or block. The declarations define items which can be used locally within the process. Note that variables may be defined here and used to store state in a model.

A process may contain a number of signal assignment statements for a given signal, which together form a *driver* for the signal. Normally there may only be one driver for a signal, and so the code which determines a signals value is confined to one process.

A process is activated initially during the initialisation phase of simulation. It executes all of the sequential statements, and then repeats, starting again with the first statement. A process may suspended itself by executing a wait statement. This is of the form:

```

wait_statement ::=
    wait [ sensitivity_clause ] [ condition_clause ] [ timeout_clause ] ;
sensitivity_clause ::= on sensitivity_list
sensitivity_list ::= signal_name { , signal_name }
condition_clause ::= until condition
timeout_clause ::= for time_expression

```

The sensitivity list of the wait statement specifies a set of signals to which the process is sensitive while it is suspended. When an event occurs

on any of these signals (that is, the value of the signal changes), the process resumes and evaluates the condition. If it is true or if the condition is omitted, execution proceeds with the next statement, otherwise the process resuspends. If the sensitivity clause is omitted, then the process is sensitive to all of the signals mentioned in the condition expression. The timeout expression must evaluate to a positive duration, and indicates the maximum time for which the process will wait. If it is omitted, the process may wait indefinitely.

If a sensitivity list is included in the header of a process statement, then the process is assumed to have an implicit wait statement at the end of its statement part. The sensitivity list of this implicit wait statement is the same as that in the process header. In this case the process may not contain any explicit wait statements.

An example of a process statements with a sensitivity list:

```
process (reset, clock)
  variable state : bit := false;
begin
  if reset then
    state := false;
  elsif clock = true then
    state := not state;
  end if;
  q <= state after prop_delay;
  -- implicit wait on reset, clock
end process;
```

During the initialization phase of simulation, the process is activated and assigns the initial value of state to the signal q. It then suspends at the implicit wait statement indicated in the comment. When either reset or clock change value, the process is resumed, and execution repeats from the beginning.

The next example describes the behaviour of a synchronization device called a Muller-C element used to construct asynchronous logic. The output of the device starts at the value '0', and stays at this value until both inputs are '1', at which time the output changes to '1'. The output then stays '1' until both inputs are '0', at which time the output changes back to '0'.

```
muller_c_2 : process
begin
  wait until a = '1' and b = '1';
  q <= '1';
  wait until a = '0' and b = '0';
  q <= '0';
end process muller_c_2 ;
```

This process does not include a sensitivity list, so explicit wait statements are used to control the suspension and activation of the process. In both wait statements, the sensitivity list is the set of signals a and b, determined from the condition expression.

4.3. Concurrent Signal Assignment Statements

Often a process describing a driver for a signal contains only one signal assignment statement. VHDL provides a convenient short-hand notation, called a concurrent signal assignment statement, for expressing such processes. The syntax is:

```

concurrent_signal_assignment_statement ::=
    [ label : ] conditional_signal_assignment
    | [ label : ] selected_signal_assignment

```

For each kind of concurrent signal assignment, there is a corresponding process statement with the same meaning.

4.3.1. Conditional Signal Assignment

A conditional signal assignment statement is a shorthand for a process containing signal assignments in an if statement. The syntax is:

```

conditional_signal_assignment ::= target <= options conditional_waveforms ;
options ::= [ guarded ] [ transport ]
conditional_waveforms ::=
    { waveform when condition else }
    waveform

```

Use of the word **guarded** is not covered in this booklet. If the word **transport** is included, then the signal assignments in the equivalent process use transport delay.

Suppose we have a conditional signal assignment:

```

s <= waveform_1 when condition_1 else
    waveform_2 when condition_2 else
    ...
    waveform_n;

```

Then the equivalent process is:

```

process
    if condition_1 then
        s <= waveform_1;
    elsif condition_2 then
        s <= waveform_2;
    elsif ...
    else
        s <= waveform_n;
    wait [ sensitivity_clause ];
end process;

```

If none of the waveform value expressions or conditions contains a reference to a signal, then the wait statement at the end of the equivalent process has no sensitivity clause. This means that after the assignment is made, the process suspends indefinitely. For example, the conditional assignment:

```

reset <= '1', '0' after 10 ns when short_pulse_required else
    '1', '0' after 50 ns;

```

schedules two transactions on the signal reset, then suspends for the rest of the simulation.

On the other hand, if there are references to signals in the waveform value expressions or conditions, then the wait statement has a sensitivity list consisting of all of the signals referenced. So the conditional assignment:

```

mux_out <= 'Z' after Tpd when en = '0' else
    in_0 after Tpd when sel = '0' else
    in_1 after Tpd;

```

is sensitive to the signals en and sel. The process is activated during the initialization phase, and thereafter whenever either of en or sel changes value.

The degenerate case of a conditional signal assignment, containing no conditional parts, is equivalent to a process containing just a signal assignment statement. So:

```
s <= waveform;
```

is equivalent to:

```
process
  s <= waveform;
  wait [ sensitivity_clause ];
end process;
```

4.3.2. Selected Signal Assignment

A selected signal assignment statement is a shorthand for a process containing signal assignments in a case statement. The syntax is:

```
selected_signal_assignment ::=
  with expression select
    target <= options selected_waveforms ;
selected_waveforms ::=
  { waveform when choices , }
  waveform when choices
choices ::= choice { | choice }
```

The options part is the same as for a conditional signal assignment. So if the word **transport** is included, then the signal assignments in the equivalent process use transport delay.

Suppose we have a selected signal assignment:

```
with expression select
  s <= waveform_1 when choice_list_1,
    waveform_2 when choice_list_2,
    ...
    waveform_n when choice_list_n;
```

Then the equivalent process is:

```
process
  case expression is
    when choice_list_1=>
      s <= waveform_1;
    when choice_list_2=>
      s <= waveform_2;
    ...
    when choice_list_n=>
      s <= waveform_n;
  end case;
  wait [ sensitivity_clause ];
end process;
```

The sensitivity list for the wait statement is determined in the same way as for a conditional signal assignment. That is, if no signals are referenced in the selected signal assignment expression or waveforms, the wait statement has no sensitivity clause. Otherwise the sensitivity clause contains all the signals referenced in the expression and waveforms.

An example of a selected signal assignment statement:

```
with alu_function select
  alu_result <= op1 + op2 when alu_add | alu_incr,
    op1 - op2 when alu_subtract,
    op1 and op2 when alu_and,
    op1 or op2 when alu_or,
    op1 and not op2 when alu_mask;
```

In this example, the value of the signal `alu_function` is used to select which signal assignment to `alu_result` to execute. The statement is sensitive to the signals `alu_function`, `op1` and `op2`, so whenever any of these change value, the selected signal assignment is resumed.

5. Model Organisation

The previous chapters have described the various facilities of VHDL somewhat in isolation. The purpose of this chapter is to show how they are all tied together to form a complete VHDL description of a digital system.

5.1. Design Units and Libraries

When you write VHDL descriptions, you write them in a *design file*, then invoke a compiler to analyse them and insert them into a *design library*. A number of VHDL constructs may be separately analysed for inclusion in a design library. These constructs are called *library units*. The *primary* library units are entity declarations, package declarations and configuration declarations (see Section 5.2). The *secondary* library units are architecture bodies and package bodies. These library units depend on the specification of their interface in a corresponding primary library unit, so the primary unit must be analysed before any corresponding secondary unit.

A design file may contain a number of library units. The structure of a design file can be specified by the syntax:

```
design_file ::= design_unit { design_unit }
design_unit ::= context_clause library_unit
context_clause ::= { context_item }
context_item ::= library_clause | use_clause
library_clause ::= library logical_name_list ;
logical_name_list ::= logical_name { , logical_name }
library_unit ::= primary_unit | secondary_unit
primary_unit ::=
    entity_declaration | configuration_declaration | package_declaration
secondary_unit ::= architecture_body | package_body
```

Libraries are referred to using identifiers called logical names. This name must be translated by the host operating system into an implementation dependent storage name. For example, design libraries may be implemented as database files, and the logical name might be used to determine the database file name. Library units in a given library can be referred to by prefixing their name with the library logical name. So for example, `ttl_lib.ttl_10` would refer to the unit `ttl_10` in library `ttl_lib`.

The context clause preceding each library unit specifies which other libraries it references and which packages it uses. The scope of the names made visible by the context clause extends until the end of the design unit.

There are two special libraries which are implicitly available to all design units, and so do not need to be named in a library clause. The first of these is called `work`, and refers to the working design library into which the

current design units will be placed by the analyser. Hence in a design unit, the previously analysed design units in a design file can be referred to using the library name work.

The second special library is called `std`, and contains the packages `standard` and `textio`. `Standard` contains all of the predefined types and functions. All of the items in this package are implicitly visible, so no `use` clause is necessary to access them.

5.2. Configurations

In Sections 3.2.3 and 3.2.4 we showed how a structural description can declare a component specification and create instances of components. We mentioned that a component declared can be thought of as a template for a design entity. The binding of an entity to this template is achieved through a configuration declaration. This declaration can also be used to specify actual generic constants for components and blocks. So the configuration declaration plays a pivotal role in organising a design description in preparation for simulation or other processing.

The syntax of a configuration declaration is:

```

configuration_declaration ::=
    configuration identifier of entity_name is
        configuration_declarative_part
        block_configuration
    end [ configuration_simple_name ] ;
configuration_declarative_part ::= { configuration_declarative_item }
configuration_declarative_item ::= use_clause
block_configuration ::=
    for block_specification
        { use_clause }
        { configuration_item }
    end for ;
block_specification ::= architecture_name / block_statement_label
configuration_item ::= block_configuration | component_configuration
component_configuration ::=
    for component_specification
        [ use binding_indication ; ]
        [ block_configuration ]
    end for ;
component_specification ::= instantiation_list : component_name
instantiation_list ::=
    instantiation_label { , instantiation_label )
    | others
    | all
binding_indication ::=
    entity_aspect
    [ generic_map_aspect ]
    [ port_map_aspect ]
entity_aspect ::=
    entity entity_name [ ( architecture_identifier ) ]
    | configuration configuration_name
    | open
generic_map_aspect ::= generic map ( generic_association_list )

```

```

entity processor is
  generic (max_clock_speed : frequency := 30 MHz);
  port ( port_list );
end processor;

architecture block_structure of processor is
  declarations
begin
  control_unit : block
    port ( port_list );
    port map ( association_list );
    declarations for control_unit
  begin
    statements for control_unit
  end block control_unit;

  data_path : block
    port ( port_list );
    port map ( association_list );
    declarations for data_path
  begin
    statements for data_path
  end block data_path;
end block_structure;

```

Figure 5-1. Example processor entity and architecture body.

```
port_map_aspect ::= port map ( port_association_list )
```

The declarative part of the configuration declaration allows the configuration to use items from libraries and packages. The outermost block configuration in the configuration declaration defines the configuration for an architecture of the named entity. For example, in Chapter 3 we had an example of a processor entity and architecture, outlined again in Figure5-1. The overall structure of a configuration declaration for this architecture might be:

```

configuration test_config of processor is
  use work.processor_types.all
  for block_structure
    configuration items
  end for;
end test_config;

```

In this example, the contents of a package called processor_types in the current working library are made visible, and the block configuration refers to the architecture block_structure of the entity processor.

Within the block configuration for the architecture, the submodules of the architecture may be configured. These submodules include blocks and component instances. A block is configured with a nested block configuration. For example, the blocks in the above architecture can be configured as shown in Figure5-2.

Where a submodule is an instance of a component, a component configuration is used to bind an entity to the component instance. To illustrate, suppose the data_path block in the above example contained an

```

configuration test_config of processor is
    use work.processor_types.all
    for block_structure
        for control_unit
            configuration items
        end for;
        for data_path
            configuration items
        end for;
    end for;
end test_config;

```

Figure5-2. Configuration of processor example.

```

data_path : block
    port ( port list );
    port map ( association list );
    component alu
        port (function : in alu_function;
            op1, op2 : in bit_vector_32;
            result : out bit_vector_32);
    end component;
    other declarations for data_path
begin
    data_alu : alu
        port map (function => alu_fn, op1 => b1, op2 => b2, result => alu_r);
    other statements for data_path
end block data_path;

```

Figure5-3. Structure of processor data-path block.

instance of the component `alu`, declared as shown in Figure5-3. Suppose also that a library `project_cells` contains an entity called `alu_cell` defined as:

```

entity alu_cell is
    generic (width : positive);
    port (function_code : in alu_function;
        operand1, operand2 : in bit_vector(width-1 downto 0);
        result : out bit_vector(width-1 downto 0);
        flags : out alu_flags);
end alu_cell;

```

with an architecture called `behaviour`. This entity matches the `alu` component template, since its operand and result ports can be constrained to match those of the component, and the flags port can be left unconnected. A block configuration for `data_path` could be specified as shown in Figure5-4.

Alternatively, if the library also contained a configuration called `alu_struct` for an architecture structure of the entity `alu_cell`, then the block configuration could use this, as shown in Figure5-5.

```

for data_path
  for data_alu : alu
    use entity project_cells.alu_cell(behaviour)
    generic map (width => 32)
    port map (function_code => function, operand1 => op1, operand2 => op2,
              result => result, flags => open);
  end for;
  other configuration items
end for;

```

Figure5-4. Block configuration using library entity.

```

for data_path
  for data_alu : alu
    use configuration project_cells.alu_struct
    generic map (width => 32)
    port map (function_code => function, operand1 => op1, operand2 => op2,
              result => result, flags => open);
  end for;
  other configuration items
end for;

```

Figure5-5. Block configuration using another configuration.

5.3. Complete Design Example

To illustrate the overall structure of a design description, a complete design file for the example in Section1.4 is shown in Figure5-6. The design file contains a number of design units which are analysed in order. The first design unit is the entity declaration of count2. Following it are two secondary units, architectures of the count2 entity. These must follow the entity declaration, as they are dependent on it. Next is another entity declaration, this being a test bench for the counter. It is followed by a secondary unit dependent on it, a structural description of the test bench. Following this is a configuration declaration for the test bench. It refers to the previously defined library units in the working library, so no library clause is needed. Notice that the count2 entity is referred to in the configuration as work.count2, using the library name. Lastly, there is a configuration declaration for the test bench using the structural architecture of count2. It uses two library units from a separate reference library, misc. Hence a library clause is included before the configuration declaration. The library units from this library are referred to in the configuration as misc.t_flipflop and misc.inverter.

This design description includes all of the design units in one file. It is equally possible to separate them into a number of files, with the opposite extreme being one design unit per file. If multiple files are used, you need to take care that you compile the files in the correct order, and re-compile dependent files if changes are made to one design unit. Source code control systems can be of use in automating this process.

```

-- primary unit: entity declaration of count2
entity count2 is
    generic (prop_delay : Time := 10 ns);
    port (clock : in bit;
         q1, q0 : out bit);
end count2;

-- secondary unit: a behavioural architecture body of count2
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;

-- secondary unit: a structural architecture body of count2
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;

```

Figure5-6. Complete design file.


```

-- primary unit: entity declaration of test bench
entity test_count2 is
end test_count2;

-- secondary unit: structural architecture body of test bench
architecture structure of test_count2 is
    signal clock, q0, q1 : bit;
    component count2
        port (clock : in bit;
              q1, q0 : out bit);
    end component;
begin
    counter : count2
        port map (clock => clock, q0 => q0, q1 => q1);

    clock_driver : process
    begin
        clock <= '0', '1' after 50 ns;
        wait for 100 ns;
    end process clock_driver;
end structure;

-- primary unit: configuration using behavioural architecture
configuration test_count2_behaviour of test_count2 is
    for structure -- of test_count2
        for counter : count2
            use entity work.count2(behaviour);
        end for;
    end for;
end test_count2_behaviour;

-- primary unit: configuration using structural architecture
library misc;
configuration test_count2_structure of test_count2 is
    for structure -- of test_count2
        for counter : count2
            use entity work.count2(structure);
            for structure -- of count_2
                for all : t_flipflop
                    use entity misc.t_flipflop(behaviour);
                end for;
                for all : inverter
                    use entity misc.inverter(behaviour);
                end for;
            end for;
        end for;
    end for;
end test_count2_structure;

```

Figure5-6 (continued).

6. Advanced VHDL

This chapter describes some more advanced facilities offered in VHDL. Although you can write many models using just the parts of the language covered in the previous chapters, you will find the features described here will significantly extend your model writing abilities.

6.1. Signal Resolution and Buses

In many digital systems, buses are used to connect a number of output drivers to a common signal. For example, if open-collector or open-drain output drivers are used with a pull-up load on a signal, the signal can be pulled low by any driver, and is only pulled high by the load when all drivers are off. This is called a *wired-or* or *wired-and* connection. On the other hand, if tri-state drivers are used, at most one driver may be active at a time, and it determines the signal value.

VHDL normally allows only one driver for a signal. (Recall that a driver is defined by the signal assignments in a process.) In order to model signals with multiple drivers, VHDL uses the notion of *resolved types* for signals. A resolved type includes in its definition a resolution function, which takes the values of all the drivers contributing to a signal, and combines them to determine the final signal value.

A resolved type for a signal is declared using the syntax for a subtype:

```
subtype_indication ::= [ resolution_function_name ] type_mark [ constraint ]
```

The resolution function name is the name of a function previously defined. The function must take a parameter which is an unconstrained array of values of the signal subtype, and must return a result of that subtype. To illustrate, consider the declarations:

```
type logic_level is (L, Z, H);
type logic_array is array (integer range <>) of logic_level;
function resolve_logic (drivers : in logic_array) return logic_level;
subtype resolved_level is resolve_logic logic_level;
```

In this example, the type `logic_level` represents three possible states for a digital signal: low (L), high-impedance (Z) and high (H). The subtype `resolved_level` can be used to declare a resolved signal of this type. The resolution function might be implemented as shown in Figure6-1.

This function iterates over the array of drivers, and if any is found to have the value L, the function returns L. Otherwise the function returns H, since all drivers are either Z or H. This models a wired-or signal with a pull-up. Note that in some cases a resolution function may be called with an empty array as the parameter, and should handle that case appropriately. The example above handles it by returning the value H, the pulled-up value.

```

function resolve_logic (drivers : in logic_array) return logic_level;
begin
  for index in drivers'range loop
    if drivers(index) = L then
      return L;
    end if;
  end loop;
  return H;
end resolve_logic;

```

Figure 7-1. Resolution function for three-state logic

6.2. Null Transactions

VHDL provides a facility to model outputs which may be turned off (for example tri-state drivers). A signal assignment may specify that no value is to be assigned to a resolved signal, that is, that the driver should be disconnected. This is done with a null waveform element. Recall that the syntax for a waveform element is:

```

waveform_element ::=
  value_expression [ after time_expression ]
  | null [ after time_expression ]

```

So an example of such a signal assignment is:

```
d_out <= null after Toz;
```

If all of the drivers of a resolved signal are disconnected, the question of the resulting signal value arises. There are two possibilities, depending on whether the signal was declared with signal kind **register** or **bus**. For register kind signals, the most recently determined value remains on the signal. This can be used to model charge storage nodes in MOS logic families. For bus kind signals, the resolution function must determine the value for the signal when no drivers are contributing to it. This is how tri-state, open-collector and open-drain buses would typically be modeled.

6.3. Generate Statements

VHDL has an additional concurrent statement which can be used in architecture bodies to describe regular structures, such as arrays of blocks, component instances or processes. The syntax is:

```

generate_statement ::=
  generate_label :
    generation_scheme generate
      { concurrent_statement }
    end generate [ generate_label ];
generation_scheme ::=
  for generate_parameter_specification
  | if condition

```

The for generation scheme describes structures which have a repeating pattern. The if generation scheme is usually used to handle exception cases within the structure, such as occur at the boundaries. This is best illustrated by example. Suppose we want to describe the structure of an

```

adder : for i in 0 to width-1 generate
    ls_bit : if i = 0 generate
        ls_cell : half_adder port map (a(0), b(0), sum(0), c_in(1));
    end generate lsbit;

    middle_bit : if i > 0 and i < width-1 generate
        middle_cell : full_adder port map (a(i), b(i), c_in(i), sum(i), c_in(i+1));
    end generate middle_bit;

    ms_bit : if i = width-1 generate
        ms_cell : full_adder port map (a(i), b(i), c_in(i), sum(i), carry);
    end generate ms_bit;
end generate adder;

```

Figure6-2. Generate statement for adder.

adder constructed out of full-adder cells, with the exception of the least significant bit, which consists of a half-adder. A generate statement to achieve this is shown in Figure6-2.

The outer generate statement iterates with *i* taking on values from 0 to width-1. For the least significant bit (*i*=0), an instance of a half adder component is generated. The input bits are connected to the least significant bits of *a* and *b*, the output bit is connected to the least significant bit of *sum*, and the carry bit is connected to the carry in of the next stage. For intermediate bits, an instance of a full adder component is generated with inputs and outputs connected similarly to the first stage. For the most significant bit (*i*=width-1), an instance of the half adder is also generated, but its carry output bit is connected to the signal *carry*.

6.4. Concurrent Assertions and Procedure Calls

There are two kinds of concurrent statement which were not covered in previous chapters: concurrent assertions and concurrent procedure calls. A concurrent assertion statement is equivalent to a process containing only an assertion statement followed by a wait statement. The syntax is:

```
concurrent_assertion_statement ::= [ label : ] assertion_statement
```

The concurrent signal assertion:

```
L : assert condition report error_string severity severity_value;
```

is equivalent to the process:

```

L : process
begin
    assert condition report error_string severity severity_value;
    wait [ sensitivity_clause ];
end process L;

```

The sensitivity clause includes all the signals which are referred to in the condition expression. If no signals are referenced, the process is activated once at simulation initialisation, checks the condition, and then suspends indefinitely.

The other concurrent statement, the concurrent procedure call, is equivalent to a process containing only a procedure call followed by a wait statement. The syntax is:

```
concurrent_procedure_call ::= [ label : ] procedure_call_statement
```

The procedure may not have any formal parameters of class **variable**, since it is not possible for a variable to be visible at any place where a concurrent statement may be used. The sensitivity list of the wait statement in the process includes all the signals which are actual parameters of mode **in** or **inout** in the procedure call. These are the only signals which can be read by the called procedure.

Concurrent procedure calls are useful for defining process behaviour that may be reused in several places or in different models. For example, suppose a package `bit_vect_arith` declares the procedure:

```
procedure add(signal a, b : in bit_vector; signal result : out bit_vector);
```

Then an example of a concurrent procedure call using this procedure is:

```
adder : bit_vect_arith.add (sample, old_accum, new_accum);
```

This would be equivalent to the process:

```
adder : process
begin
    bit_vect_arith.add (sample, old_accum, new_accum);
    wait on sample, old_accum;
end process adder;
```

6.5. Entity Statements

In Section 3.1, it was mentioned that an entity declaration may include statements for monitoring operation of the entity. Recall that the syntax for an entity declaration is:

```
entity_declaration ::=
    entity identifier is
        entity_header
        entity_declarative_part
    [ begin
        entity_statement_part ]
    end [ entity_simple_name ] ;
```

The syntax for the statement part is:

```
entity_statement_part ::= { entity_statement }
entity_statement ::=
    concurrent_assertion_statement
    / passive_concurrent_procedure_call
    / passive_process_statement
```

The concurrent statement that are allowed in an entity declaration must be *passive*, that is, they may not contain any signal assignments. (This includes signal assignments inside nested procedures of a process.) A result of this rule is that such processes cannot modify the state of the entity, or any circuit the entity may be used in. However, they can fully monitor the state, and so may be used to report erroneous operating conditions, or to trace the behavior of the design.