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

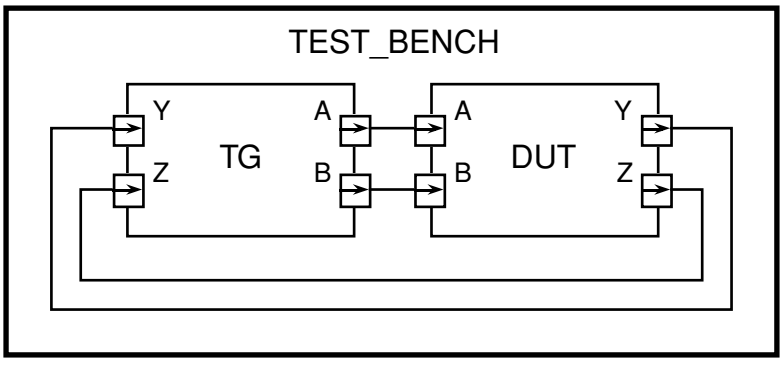

Figure 3-1. Test bench circuit.

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 Figure 3-1. 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 Chapter 4.

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 Section 1.4. 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;
```

Figure 3-2. 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 Section 3.3) 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.