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 sytems, 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 is 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 connectected 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 Section3.1, 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.