~~Section~~ Clause[1] 9

Concurrent statements

The various forms of concurrent statements are described in this ~~section~~ clause[2]. Concurrent statements are used to define interconnected blocks and processes that jointly describe the overall behavior or structure of a design. Concurrent statements execute asynchronously with respect to each other.

concurrent_statement ::=
    block_statement
  | process_statement
  | concurrent_procedure_call_statement
  | concurrent_assertion_statement
  | concurrent_signal_assignment_statement
  | component_instantiation_statement
  | generate_statement

The primary concurrent statements are the block statement, which groups together other concurrent statements, and the process statement, which represents a single independent sequential process. Additional concurrent statements provide convenient syntax for representing simple, commonly occurring forms of processes, as well as for representing structural decomposition and regular descriptions.

Within a given simulation cycle, an implementation may execute concurrent statements in parallel or in some order. The language does not define the order, if any, in which such statements will be executed. A description that depends upon a particular order of execution of concurrent statements is erroneous.

All concurrent statements may be labeled. Such labels are implicitly declared at the beginning of the declarative part of the innermost enclosing entity declaration, architecture body, block statement, or generate statement.

9.1 Block statement

A block statement defines an internal block representing a portion of a design. Blocks may be hierarchically nested to support design decomposition.

block_statement ::=
    *block*_label :
        block [(*guard*_expression)] [**is**]
            block_header
            block_declarative_part
        begin
            block_statement_part
        end block [*block*_label] ;

1. To conform to IEEE rules.
2. To conform to IEEE rules.

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

block_declarative_part ::=
    { block_declarative_item }

block_statement_part ::=
    { concurrent_statement }
```

If a guard expression appears after the reserved word **block**, then a signal with the simple name GUARD of predefined type BOOLEAN is implicitly declared at the beginning of the declarative part of the block, and the guard expression defines the value of that signal at any given time (see 12.6.4). The type of the guard expression must be type BOOLEAN. Signal GUARD may be used to control the operation of certain statements within the block (see 9.5).

The implicit signal GUARD must not have a source.

If a block header appears in a block statement, it explicitly identifies certain values or signals that are to be imported from the enclosing environment into the block and associated with formal generics or ports. The generic and port clauses define the formal generics and formal ports of the block (see 1.1.1.1 and 1.1.1.2); the generic map and port map aspects define the association of actuals with those formals (see 5.2.1.2). Such actuals are evaluated in the context of the enclosing declarative region.

If a label appears at the end of a block statement, it must repeat the block label.

NOTES

1—The value of signal GUARD is always defined within the scope of a given block, and it does not implicitly extend to design entities bound to components instantiated within the given block. However, the signal GUARD may be explicitly passed as an actual signal in a component instantiation in order to extend its value to lower-level components.

2—An actual appearing in a port association list of a given block can never denote a formal port of the same block.

9.2 Process statement

A process statement defines an independent sequential process representing the behavior of some portion of the design.

```
process_statement ::=
    [ process_label : ]
        [ postponed ] process [ ( sensitivity_list ) ] [ is ]
            process_declarative_part
        begin
            process_statement_part
        end [ postponed ] process [ process_label ] ;

process_declarative_part ::=
    { process_declarative_item }
```

process_declarative_item ::=
    subprogram_declaration
  | subprogram_body
  | type_declaration
  | subtype_declaration
  | constant_declaration
  | variable_declaration
  | file_declaration
  | alias_declaration
  | attribute_declaration
  | attribute_specification
  | use_clause
  | ~~group_type_declaration~~ group_template_declaration[3]
  | group_declaration

process_statement_part ::=
  { sequential_statement }

If the reserved word **postponed** precedes the initial reserved word **process**, the process statement defines a *postponed process*; otherwise, the process statement defines a *nonpostponed process*.

If a sensitivity list appears following the reserved word **process**, then the process statement is assumed to contain an implicit wait statement as the last statement of the process statement part; this implicit wait statement is of the form

wait on sensitivity_list ;

where the sensitivity list of the wait statement is that following the reserved word **process**. Such a process statement must not contain an explicit wait statement. Similarly, if such a process statement is a parent of a procedure, then it is an error if[4] that procedure ~~may not contain~~ contains[5] a wait statement.

~~Only static signal names (see 6.1) for which reading is permitted may appear~~ It is an error if any name that does not denote a static signal name (see 6.1) for which reading is permitted appears[6] in the sensitivity list of a process statement.

If the reserved word **postponed** appears at the end of a process statement, the process must be a postponed process. If a label appears at the end of a process statement, the label must repeat the process label.

It is an error if a variable declaration in a process declarative part declares a shared variable.

The execution of a process statement consists of the repetitive execution of its sequence of statements. After the last statement in the sequence of statements of a process statement is executed, execution will immediately continue with the first statement in the sequence of statements.

A process statement is said to be a *passive process* if neither the process itself, nor any procedure of which the process is a parent, contains a signal assignment statement. ~~Such a process, or any concurrent statement equivalent to such a process, may appear~~ It is an error if a process or a concurrent statement, other than a passive process or a concurrent statement equivalent to such a process, appears[7] in the entity statement part of an entity declaration.

3. IR1000.2.11.
4. IR1000.4.7.
5. IR1000.4.7.
6. IR1000.4.7.
7. IR1000.4.7.

NOTES

1—The above rules imply that a process that has an explicit sensitivity list always has exactly one (implicit) wait statement in it, and that wait statement appears at the end of the sequence of statements in the process statement part. Thus, a process with a sensitivity list always waits at the end of its statement part; any event on a signal named in the sensitivity list will cause such a process to execute from the beginning of its statement part down to the end, where it will wait again. Such a process executes once through at the beginning of simulation, suspending for the first time when it executes the implicit wait statement.

2—The time at which a process executes after being resumed by a wait statement (see 8.1) differs depending on whether the process is postponed or nonpostponed. When a nonpostponed process is resumed, it executes in the current simulation cycle (see 2.6.4). When a postponed process is resumed, it does not execute until a simulation cycle occurs in which the next simulation cycle is not a delta cycle. In this way, a postponed process accesses the values of signals that are the "final" values at the current simulated time.

3—The conditions that cause a process to resume execution may no longer hold at the time the process resumes execution if the process is a postponed process.

9.3 Concurrent procedure call statements

A concurrent procedure call statement represents a process containing the corresponding sequential procedure call statement.

concurrent_procedure_call_statement ::=
    [label :] [**postponed**] procedure_call ;

For any concurrent procedure call statement, there is an equivalent process statement. The equivalent process statement is a postponed process if and only if the concurrent procedure call statement includes the reserved word **postponed**. The equivalent process statement has a label if and only if the concurrent procedure call statement has a label; if the equivalent process statement has a label, it is the same as that of the concurrent procedure call statement. The equivalent process statement also has no sensitivity list, an empty declarative part, and a statement part that consists of a procedure call statement followed by a wait statement.

The procedure call statement consists of the same procedure name and actual parameter part that appear in the concurrent procedure call statement.

If there exists a name that denotes a signal in the actual part of any association element in the concurrent procedure call statement, and that actual is associated with a formal parameter of mode **in** or **inout**, then the equivalent process statement includes a final wait statement with a sensitivity clause that is constructed by taking the union of the sets constructed by applying the rule of 8.1 to each actual part associated with a formal parameter.

Execution of a concurrent procedure call statement is equivalent to execution of the equivalent process statement.

Example:

```
CheckTiming (tPLH, tPHL, Clk, D, Q);          -- A concurrent procedure call statement.

process                                       -- The equivalent process.
begin
    CheckTiming (tPLH, tPHL, Clk, D, Q);
    wait on Clk, D, Q;
end process;
```

NOTES

1—Concurrent procedure call statements make it possible to declare procedures representing commonly used processes and to create such processes easily by merely calling the procedure as a concurrent statement. The wait statement at the end of the statement part of the equivalent process statement allows a procedure to be called without having it loop interminably, even if the procedure is not necessarily intended for use as a process (i.e., it contains no wait statement). Such a procedure

may persist over time (and thus the values of its variables ~~may~~[8] retain state over time) if its outermost statement is a loop statement and the loop contains a wait statement. Similarly, such a procedure may be guaranteed to execute only once, at the beginning of simulation, if its last statement is a wait statement that has no sensitivity clause, condition clause, or timeout clause.

2—The value of an implicitly declared signal GUARD has no effect on evaluation of a concurrent procedure call unless it is explicitly referenced in one of the actual parts of the actual parameter part of the concurrent procedure call statement.

9.4 Concurrent assertion statements

A concurrent assertion statement represents a passive process statement containing the specified assertion statement.

concurrent_assertion_statement ::=
[label :] [**postponed**] assertion ;

For any concurrent assertion statement, there is an equivalent process statement. The equivalent process statement is a postponed process if and only if the concurrent assertion statement includes the reserved word **postponed**. The equivalent process statement has a label if and only if the concurrent assertion statement has a label; if the equivalent process statement has a label, it is the same as that of the concurrent assertion statement. The equivalent process statement also has no sensitivity list, an empty declarative part, and a statement part that consists of an assertion statement followed by a wait statement.

The assertion statement consists of the same condition, **report** clause, and **severity** clause that appear in the concurrent assertion statement.

If there exists a name that denotes a signal in the Boolean expression that defines the condition of the assertion, then the equivalent process statement includes a final wait statement with a sensitivity clause that is constructed by applying the rule of 8.1 to that expression; otherwise, the equivalent process statement contains a final wait statement that has no explicit sensitivity clause, condition clause, or timeout clause.

Execution of a concurrent assertion statement is equivalent to execution of the equivalent process statement.

NOTES

1—Since a concurrent assertion statement represents a passive process statement, such a process has no outputs. Therefore, the execution of a concurrent assertion statement will never cause an event to occur. However, if the assertion is false, then the specified error message will be sent to the simulation report.

2—The value of an implicitly declared signal GUARD has no effect on evaluation of the assertion unless it is explicitly referenced in one of the expressions of that assertion.

3—A concurrent assertion statement whose condition is defined by a static expression is equivalent to a process statement that ends in a wait statement that has no sensitivity clause; such a process will execute once through at the beginning of simulation and then wait indefinitely.

9.5 Concurrent signal assignment statements

A concurrent signal assignment statement represents an equivalent process statement that assigns values to signals.

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

options ::= [**guarded**] [delay_mechanism]

8. IR1000.4.7.

There are two forms of the concurrent signal assignment statement. For each form, the characteristics that distinguish it are discussed in the following paragraphs.

Each form may include one or both of the two options **guarded** and a delay mechanism (see 8.4 for the delay mechanism, 9.5.1 for the conditional signal assignment statement, and 9.5.2 for the selected signal assignment statement). The option **guarded** specifies that the signal assignment statement is executed when a signal GUARD changes from FALSE to TRUE, or when that signal has been TRUE and an event occurs on one of the signal assignment statement's inputs. (The signal GUARD ~~may be~~ must be either[9] one of the implicitly declared GUARD signals associated with block statements that have guard expressions, or it ~~may~~ must[10] be an explicitly declared signal of type Boolean that is visible at the point of the concurrent signal assignment statement.) The delay mechanism option specifies the pulse rejection characteristics of the signal assignment statement.

If the target of a concurrent signal assignment is a name that denotes a guarded signal (see 4.3.1.2), or if it is in the form of an aggregate and the expression in each element association of the aggregate is a static signal name denoting a guarded signal, then the target is said to be a *guarded target*. If the target of a concurrent signal assignment is a name that denotes a signal that is not a guarded signal, or if it is in the form of an aggregate and the expression in each element association of the aggregate is a static signal name denoting a signal that is not a guarded signal, then the target is said to be an *unguarded target*. It is an error if the target of a concurrent signal assignment is neither a guarded target nor an unguarded target.

For any concurrent signal assignment statement, there is an equivalent process statement with the same meaning. The process statement equivalent to a concurrent signal assignment statement whose target is a signal name is constructed as follows:

a) If a label appears on the concurrent signal assignment statement, then the same label appears on the process statement.

b) The equivalent process statement is a postponed process if and only if the concurrent signal assignment statement includes the reserved word **postponed**.

c) If the delay mechanism option appears in the concurrent signal assignment, then the same delay mechanism appears in every signal assignment statement in the process statement; otherwise, it appears in no signal assignment statement in the process statement.

d) The statement part of the equivalent process statement consists of a statement transform (described below).

   If the option **guarded** appears in the concurrent signal assignment statement, then the concurrent signal assignment is called a *guarded assignment*. If the concurrent signal assignment statement is a guarded assignment, and if the target of the concurrent signal assignment is a guarded target, then the statement transform is as follows:

```
if GUARD then
    signal_transform
else
    disconnection_statements
end if ;
```

   Otherwise, if the concurrent signal assignment statement is a guarded assignment, but if the target of the concurrent signal assignment is *not* a guarded target, then the statement transform is as follows:

```
if GUARD then
    signal_transform
end if ;
```

9. IR1000.4.7.
10. IR1000.4.7.

Finally, if the concurrent signal assignment statement is *not* a guarded assignment, and if the target of the concurrent signal assignment is *not* a guarded target, then the statement transform is as follows:

signal_transform

It is an error if a concurrent signal assignment is not a guarded assignment and the target of the concurrent signal assignment is a guarded target.

A *signal transform* is either a sequential signal assignment statement, an if statement, a case statement, or a null statement.If the signal transform is an if statement or a case statement, then it contains either sequential signal assignment statements or null statements, one for each of the alternative waveforms. The signal transform determines which of the alternative waveforms is to be assigned to the output signals.

e) If the concurrent signal assignment statement is a guarded assignment, or if any expression (other than a time expression) within the concurrent signal assignment statement references a signal, then the process statement contains a final wait statement with an explicit sensitivity clause. The sensitivity clause is constructed by taking the union of the sets constructed by applying the rule of 8.1 to each of the aforementioned expressions. Furthermore, if the concurrent signal assignment statement is a guarded assignment, then the sensitivity clause also contains the simple name GUARD. (The signals identified by these names are called the *inputs* of the signal assignment statement.) Otherwise, the process statement contains a final wait statement that has no explicit sensitivity clause, condition clause, or timeout clause.

Under certain conditions (see above) the equivalent process statement may contain a sequence of disconnection statements. A *disconnection statement* is a sequential signal assignment statement that assigns a null transaction to its target. If a sequence of disconnection statements is present in the equivalent process statement, the sequence consists of one sequential signal assignment for each scalar subelement of the target of the concurrent signal assignment statement. For each such sequential signal assignment, the target of the assignment is the corresponding scalar subelement of the target of the concurrent signal assignment, and the waveform of the assignment is a null waveform element whose time expression is given by the applicable disconnection specification (see 5.3).

If the target of a concurrent signal assignment statement is in the form of an aggregate, then the same transformation applies. Such a target ~~may only contain~~ must contain only[11] locally static signal names, ~~and a signal may not be~~ ; moreover, it is an error if any signal is[12] identified by more than one signal name.

It is an error if a null waveform element appears in a waveform of a concurrent signal assignment statement.

Execution of a concurrent signal assignment statement is equivalent to execution of the equivalent process statement.

NOTES

1—A concurrent signal assignment statement whose waveforms and target contain only static expressions is equivalent to a process statement whose final wait statement has no explicit sensitivity clause, so it will execute once through at the beginning of simulation and then suspend permanently.

2—A concurrent signal assignment statement whose waveforms are all the reserved word **unaffected** has no drivers for the target, since every waveform in the concurrent signal assignment statement is transformed to the statement

null;

in the equivalent process statement. See 9.5.1.

11. IR1000.4.7.
12. IR1000.4.7.

9.5.1 Conditional signal assignments

The conditional signal assignment represents a process statement in which the signal transform is an if statement.

```
conditional_signal_assignment ::=
    target  <=  options conditional_waveforms ;

conditional_waveforms ::=
    { waveform when condition else }
      waveform [ when condition ]
```

The options for a conditional signal assignment statement are discussed in 9.5.

For a given conditional signal assignment, there is an equivalent process statement corresponding to it as defined for any concurrent signal assignment statement. If the conditional signal assignment is of the form

```
target <= options
    waveform1 when condition1 else
    waveform2 when condition2 else
    • • •
    waveformN–1 when conditionN–1 else
    waveformN when conditionN;
```

then the signal transform in the corresponding process statement is of the form

```
if condition1 then
    wave_transform1
elsif condition2 then
    wave_transform2
• • •
elsif conditionN–1 then
    wave_transformN–1
elsif conditionN then
    wave_transformN
end if ;
```

If the conditional waveform is only a single waveform, the signal transform in the corresponding process statement is of the form

```
wave_transform
```

For any waveform, there is a corresponding *wave transform*. If the waveform is of the form

```
waveform_element1, waveform_element2, …, waveform_elementN
```

then the wave transform in the corresponding process statement is of the form

```
target <= [ delay_mechanism ] waveform_element1, waveform_element2, …,
                                        waveform_elementN;
```

If the waveform is of the form

```
unaffected
```

then the wave transform in the corresponding process statement is of the form

```
null;
```

In this example, the final **null** causes the driver to be unchanged, rather than disconnected. (This is the null statement—not a null waveform element).

The characteristics of the waveforms and conditions in the conditional assignment statement must be such that the if statement in the equivalent process statement is a legal statement.

Example:

```
S <= unaffected when Input_pin = S'DrivingValue else
     Input_pin after Buffer_Delay;
```

NOTE

—The wave transform of a waveform of the form **unaffected** is the null statement, not the null transaction.

9.5.2 Selected signal assignments

The selected signal assignment represents a process statement in which the signal transform is a case statement.

```
selected_signal_assignment ::=
     with expression select
          target <= options selected_waveforms ;

selected_waveforms ::=
     { waveform when choices , }
       waveform when choices
```

The options for a selected signal assignment statement are discussed in 9.5.

For a given selected signal assignment, there is an equivalent process statement corresponding to it as defined for any concurrent signal assignment statement. If the selected signal assignment is of the form

```
with expression select
     target <= options     waveform1      when choice_list1 ,
                           waveform2      when choice_list2 ,
                           • • •
                           waveformN–1    when choice_listN–1,
                           waveformN      when choice_listN ;
```

then the signal transform in the corresponding process statement is of the form

```
case expression is
     when choice_list1 =>
          wave_transform1
     when choice_list2 =>
          wave_transform2
     • • •
     when choice_listN–1 =>
          wave_transformN–1
     when choice_listN =>
          wave_transformN
end case ;
```

Wave transforms are defined in 9.5.1.

The characteristics of the select expression, the waveforms, and the choices in the selected assignment statement must be such that the case statement in the equivalent process statement is a legal statement.

9.6 Component instantiation statements

A component instantiation statement defines a subcomponent of the design entity in which it appears, associates signals or values with the ports of that subcomponent, and associates values with generics of that subcomponent. This subcomponent is one instance of a class of components defined by a corresponding component declaration, design entity, or configuration declaration.

```
component_instantiation_statement ::=
    instantiation_label :
        instantiated_unit
            [ generic_map_aspect ]
            [ port_map_aspect ] ;

instantiated_unit ::=
     [ component ] component_name
    | entity entity_name [ ( architecture_identifier ) ]
    | configuration configuration_name
```

The component name, if present, must be the name of a component declared in a component declaration. The entity name, if present, must be the name of a previously analyzed entity ~~interface~~ declaration[13]; if an architecture identifier appears in the instantiated unit, then that identifier must be the same as the simple name of an architecture body associated with the entity declaration denoted by the corresponding entity name. The architecture identifier defines a simple name that is used during the elaboration of a design hierarchy to select the appropriate architecture body. The configuration name, if present, must be the name of a previously analyzed configuration declaration. The generic map aspect, if present, optionally associates a single actual with each local generic (or member thereof) in the corresponding component declaration or entity ~~interface~~ declaration[14]. Each local generic (or member thereof) must be associated at most once. Similarly, the port map aspect, if present, optionally associates a single actual with each local port (or member thereof) in the corresponding component declaration or entity ~~interface~~ declaration[15]. Each local port (or member thereof) must be associated at most once. The generic map and port map aspects are described in 5.2.1.2.

If an instantiated unit containing the reserved word **entity** does not contain an explicitly specified architecture identifier, then the architecture identifier is implicitly specified according to the rules given in 5.2.2. The architecture identifier defines a simple name that is used during the elaboration of a design hierarchy to select the appropriate architecture body.

A component instantiation statement and a corresponding configuration specification, if any, taken together, imply that the block hierarchy within the design entity containing the component instantiation is to be extended with a unique copy of the block defined by another design entity. The generic map and port map aspects in the component instantiation statement and in the binding indication of the configuration specification identify the connections that are to be made in order to accomplish the extension.

NOTES

1—A configuration specification can be used to bind a particular instance of a component to a design entity and to associate the local generics and local ports of the component with the formal generics and formal ports of that design entity. A configuration specification ~~may~~ can[16] apply to a component instantiation statement only if the name in the instantiated unit of the component instantiation statement denotes a component declaration. (See 5.2.)

2—The component instantiation statement may be used to imply a structural organization for a hardware design. By using component declarations, signals, and component instantiation statements, a given (internal or external) block may be described in terms of subcomponents that are interconnected by signals.

---

13. Terminological correction.
14. Terminological correction.
15. Terminological correction.
16. IR1000.4.7.

3—Component instantiation provides a way of structuring the logical decomposition of a design. The precise structural or behavioral characteristics of a given subcomponent may be described later, provided that the instantiated unit is a component declaration. Component instantiation also provides a mechanism for reusing existing designs in a design library. A configuration specification can bind a given component instance to an existing design entity, even if the generics and ports of the entity declaration do not precisely match those of the component (provided that the instantiated unit is a component declaration); if the generics or ports of the entity declaration do not match those of the component, the configuration specification must contain a generic map or port map, as appropriate, to map the generics and ports of the entity declaration to those of the component.

9.6.1 Instantiation of a component

A component instantiation statement whose instantiated unit contains a name denoting a component is equivalent to a ~~pair~~ triple[17] of nested block statements that couple the block hierarchy in the containing design unit to a unique copy of the block hierarchy contained in another design unit (i.e., the subcomponent). The outer block represents the component declaration; the ~~inner~~ intermediate[18] block represents the ~~design~~ entity declaration[19] to which the component is bound; and the inner block represents the corresponding architecure body[20]. Each is defined by a block statement.

The header of the block statement corresponding to the component declaration consists of the generic and port clauses (if present) that appear in the component declaration, followed by the generic map and port map aspects (if present) that appear in the corresponding component instantiation statement. The meaning of any identifier appearing in the header of this block statement is associated with the corresponding occurrence of the identifier in the generic clause, port clause, generic map aspect, or port map aspect, respectively. The statement part of the block statement corresponding to the component declaration consists of a nested block statement corresponding to the ~~design~~[21] entity declaration[22].

The header of the block statement corresponding to the ~~design~~[23] entity declaration[24] consists of the generic and port clauses (if present) that appear in the entity declaration ~~that defines the interface to the design entity~~[25], followed by the generic map and port map aspects (if present) that appear in the binding indication that binds the component instance to that ~~design~~[26] entity declaration[27]. The declarative part of the block statement corresponding to the ~~design~~[28] entity declaration[29] consists of the declarative items from the entity declarative part~~, followed by the declarative items from the declarative part of the corresponding architecture body~~[30]. The statement part of the block statement corresponding to the ~~design~~[31] entity declaration[32] consists of the concurrent statements from the entity statement part, followed by ~~the concurrent statements from the statement part of~~ a nested block statement corresponding to[33] the corresponding architecture body. The meaning of any identifier appearing anywhere in this intermediate[34] block statement is that associated with the corresponding occurrence of the identifier in the entity declaration ~~or architecture body, respectively~~[35].

17. LCS 3.
18. LCS 3.
19. LCS 3.
20. LCS 3.
21. LCS 3.
22. LCS 3.
23. LCS 3.
24. LCS 3.
25. LCS 3.
26. LCS 3.
27. LCS 3.
28. LCS 3.
29. LCS 3.
30. LCS 3.
31. LCS 3.
32. LCS 3.
33. LCS 3.
34. LCS 3.
35. LCS 3.

The header of the block statement corresponding to the architecture body is empty. The declarative part of the block statement corresponding to the architecture body consists of the declarative items from the declarative part of the corresponding architecture body. The statement part of the block statement corresponding to the architecture body consists of the concurrent statements from the statement part of the corresponding architecture body. The meaning of any identifier appearing anywhere in this block statement is that associated with the corresponding occurrence of the identifier in the architecture body.[36]

For example, consider the following component declaration, instantiation, and corresponding configuration specification:

```
component
    COMP port (A,B : inout BIT);
end component;

for C: COMP use
    entity X(Y)
    port map (P1 => A, P2 => B) ;
            • • •
C:  COMP port map (A => S1, B => S2);
```

Given the following entity declaration and architecture declaration:

```
entity X is
    port (P1, P2 : inout BIT);
    constant Delay: Time := 1 ms;
begin
    CheckTiming (P1, P2, 2*Delay);
end X ;

architecture Y of X is
    signal P3: Bit;
begin
    P3 <= P1 after Delay;
    P2 <= P3 after Delay;
    B:  block
            • • •
        begin
            • • •
        end block;
end Y;
```

then the following block statements implement the coupling between the block hierarchy in which component instantiation statement C appears and the block hierarchy contained in design entity X(Y):

```
C: block                                        -- Component block.
        port (A,B : inout BIT);                 -- Local ports.
        port map (A => S1, B => S2);            -- Actual/local binding.
    begin
        X:  block                               -- Design entity block.
                port (P1, P2 : inout BIT);      -- Formal ports.
                port map (P1 => A, P2 => B);    -- Local/formal binding.
                constant Delay: Time := 1 ms;   -- Entity declarative item.
                ~~signal P3: Bit;~~             ~~-- Architecture declarative item.~~[37]
            begin
```

36. LCS 3.
37. LCS 3.

```
        CheckTiming (P1, P2, 2*Delay);          -- Entity statement.
        Y: block
            signal P3: Bit;                     -- Architecture declarative item
        begin[38]
            P3 <= P1 after Delay;               -- Architecture statements.
            P2 <= P3 after Delay;
            B: block                            -- Internal block hierarchy.
                • • •
            begin
                • • •
            end block;
        end block Y;[39]
    end block X ;
end block C;
```

The block hierarchy extensions implied by component instantiation statements that are bound to design entities are accomplished during the elaboration of a design hierarchy (see ~~Section~~ Clause[40] 12).

9.6.2 Instantiation of a design entity

A component instantiation statement whose instantiated unit denotes either a design entity or a configuration declaration is equivalent to a ~~pair~~ triple[41] of nested block statements that couple the block hierarchy in the containing design unit to a unique copy of the block hierarchy contained in another design unit (i.e., the subcomponent). The outer block represents the component instantiation statement; the ~~inner~~ intermediate[42] block represents the ~~design~~[43] entity declaration[44] to which the instance is bound; and the inner block represents the corresponding architecture body[45]. Each is defined by a block statement.

The header of the block statement corresponding to the component instantiation statement is empty, as is the declarative part of this block statement. The statement part of the block statement corresponding to the component declaration consists of a nested block statement corresponding to the ~~design~~[46] entity declaration[47].

The header of the block statement corresponding to the ~~design~~[48] entity declaration[49] consists of the generic and port clauses (if present) that appear in the entity declaration that defines the interface to the design entity, followed by the generic map and port map aspects (if present) that appear in the component instantiation statement that binds the component instance to a copy of that design entity. The declarative part of the block statement corresponding to the ~~design~~[50] entity declaration[51] consists of the declarative items from the entity declarative part~~, followed by the declarative items from the declarative part of the corresponding architecture body~~[52]. The statement part of the block statement corresponding to the ~~design~~[53] entity declaration[54] consists of the concurrent statements from the entity statement part, followed by ~~the concurrent statements from the statement part of~~ a nested block statement corresponding to[55] the corresponding architecture body. The meaning of any identifier appearing any-

38. LCS 3.
39. LCS 3.
40. To conform to IEEE rules.
41. LCS 3.
42. LCS 3.
43. LCS 3.
44. LCS 3.
45. LCS 3.
46. LCS 3.
47. LCS 3.
48. LCS 3.
49. LCS 3.
50. LCS 3.
51. LCS 3.
52. LCS 3.
53. LCS 3.
54. LCS 3.

where in this block statement is that associated with the corresponding occurrence of the identifier in the entity declaration ~~or architecture body, respectively~~[56].

The header of the block statement corresponding to the architecture body is empty. The declarative part of the block statement corresponding to the architecture body consists of the declarative items from the declarative part of the corresponding architecture body. The statement part of the block statement corresponding to the architecture body consists of the concurrent statements from the statement part of the corresponding architecture body. The meaning of any identifier appearing anywhere in this block statement is that associated with the corresponding occurrence of the identifier in the architecture body.[57]

For example, consider the following design entity:

```
entity X is
    port (P1, P2: inout BIT);
    constant Delay: DELAY_LENGTH := 1 ms;
    use WORK.TimingChecks.all;
begin
    CheckTiming (P1, P2, 2*Delay);
end entity X;

architecture Y of X is
    signal P3: BIT;
begin
    P3 <= P1 after Delay;
    P2 <= P3 after Delay;
    B: block
        • • •
    begin
        • • •
    end block B;
end architecture Y;
```

This design entity is instantiated by the following component instantiation statement:

```
C: entity Work.X (Y) port map (P1 => S1, P2 => S2);
```

The following block statements implement the coupling between the block hierarchy in which component instantiation statement C appears and the block hierarchy contained in design entity X(Y):

```
C: block                                          -- Instance block.
begin
    X: block                                      -- Design entity block.
        port (P1, P2: inout BIT);                 -- Entity ~~interface~~ declaration[58] ports.
        port map (P1 => S1, P2 => S2);            -- Instantiation statement port map.
        constant Delay: DELAY_LENGTH := 1 ms;     -- Entity declarative items.
        use WORK.TimingChecks.all;
        ~~signal P3: BIT;~~                       ~~-- Architecture declarative item.~~[59]
    begin
        CheckTiming (P1, P2, 2*Delay);            -- Entity statement.
        Y: block
            signal P3: BIT;                       -- Architecture declarative item.
```

55. LCS 3.
56. LCS 3.
57. LCS 3.
58. Terminological correction.
59. LCS 3.

```
        begin[60]
            P3 <= P1 after Delay;                    -- Architecture statements.
            P2 <= P3 after Delay;
            B: block
                • • •
            begin
                • • •
            end block B;
        end block Y;[61]
    end block X;
end block C;
```

Moreover, consider the following design entity, which is followed by an associated configuration declaration and component instantiation:

```
entity X is
    port (P1, P2: inout BIT);
    constant Delay: DELAY_LENGTH := 1 ms;
    use WORK.TimingChecks.all;
begin
    CheckTiming (P1, P2, 2*Delay);
end entity X;

architecture Y of X is
    signal P3: BIT;
begin
    P3 <= P1 after Delay;
    P2 <= P3 after Delay;
    B: block
        • • •
    begin
        • • •
    end block B;
end architecture Y;
```

The configuration declaration is

```
configuration Alpha of X is
    for Y
        • • •
    end for;
end configuration Alpha;
```

The component instantiation is

```
C: configuration Work.Alpha port map (P1 => S1, P2 => S2);
```

The following block statements implement the coupling between the block hierarchy in which component instantiation statement C appears and the block hierarchy contained in design entity X(Y):

60. LCS 3.
61. LCS 3.

```
C: block                                          -- Instance block.
begin
    X: block                                      -- Design entity block.
        port (P1, P2: inout BIT);                 -- Entity ~~interface~~ declaration[62] ports.
        port map (P1 => S1, P2 => S2);            -- Instantiation statement port map.
        constant Delay: DELAY_LENGTH := 1 ms;     -- Entity declarative items.
        use WORK.TimingChecks.all;
        ~~signal P3: BIT;~~                       ~~-- Architecture declarative item.~~[63]
    begin
        CheckTiming (P1, P2, 2*Delay);            -- Entity statement.
        Y: begin
            signal P3: BIT:                       -- Architecture declarative item.
        begin[64]
            P3 <= P1 after Delay;                 -- Architecture statements.
            P2 <= P3 after Delay;
            B: block
                • • •
            begin
                • • •
            end block B;
        end block Y;[65]
    end block X;
end block C;
```

The block hierarchy extensions implied by component instantiation statements that are bound to design entities occur during the elaboration of a design hierarchy (see ~~Section~~ Clause[66] 12).

9.7 Generate statements

A generate statement provides a mechanism for iterative or conditional elaboration of a portion of a description.

```
generate_statement ::=
    generate_label :
        generation_scheme generate
            [ { block_declarative_item }
        begin ]
            { concurrent_statement }
        end generate [ generate_label ] ;

generation_scheme ::=
      for generate_parameter_specification
    | if condition

label ::= identifier
```

If a label appears at the end of a generate statement, it must repeat the generate label.

For a generate statement with a **for** generation scheme, the generate parameter specification is the declaration of the *generate parameter* with the given identifier. The generate parameter is a constant object whose type is the base type of the discrete range of the generate parameter specification.

---

62. Terminological correction.
63. LCS 3.
64. LCS 3.
65. LCS 3.
66. To conform to IEEE rules.

The discrete range in a generation scheme of the first form must be a static discrete range; similarly, the condition in a generation scheme of the second form must be a static expression.

The elaboration of a generate statement is described in 12.4.2.

Example:

```
Gen: block
     begin
          L1: CELL port map (Top, Bottom, A(0), B(0)) ;

          L2: for I in 1 to 3 generate
               L3: for J in 1 to 3 generate
                    L4: if I+J>4 generate
                         L5: CELL port map (A(I–1),B(J–1),A(I),B(J)) ;
                    end generate ;
               end generate ;
          end generate ;

            L6: for I in 1 to 3 generate
                      L7: for J in 1 to 3 generate
                                L8: if I+J<4 generate
                                          L9: CELL port map (A(I+1),B(J+1),A(I),B(J)) ;
                                end generate ;
                      end generate ;
            end generate ;
   end block Gen;
```