6. SEQUENTIAL AND CONCURRENT STATEMENTS IN THE VHDL LANGUAGE

A VHDL description has two domains: a sequential domain and a concurrent domain. The *sequential* domain is represented by a process or subprogram that contains sequential statements. These statements are executed in the order in which they appear within the process or subprogram, as in programming languages. The *concurrent* domain is represented by an architecture that contains processes, concurrent procedure calls, concurrent signal assignments, and component instantiations (described in Laboratory No. 8).

This laboratory work presents the format and use of sequential and concurrent statements. As examples some basic combinational and sequential circuits are described, such as multiplexers, decoders, flip-flops, registers, and counters.

6.1. Sequential Statements

In this section, first some aspects related to processes are described, such as process specification, process execution, `wait` statement, and the difference between combinational and sequential processes. Then the sequential statements that may appear in a process or subprogram are presented: sequential signal assignment, variable assignment, `if` statement, `case` statement, loop statements (`loop`, `while loop`, `for loop`, `next`, `exit`), and the sequential `assert` statement. Besides these statements, other sequential statements are the procedure call statement and the return statement from a procedure or function. These latter statements are presented in Laboratory No. 9 which describes subprograms.

6.1.1. Processes

A process is a sequence of statements that are executed in the specified order. The process declaration delimits a sequential domain of the architecture in which the declaration appears. Processes are used for behavioral descriptions.

6.1.1.1. Structure and Execution of a Process

A process may appear anywhere in an architecture body (the part starting after the `begin` keyword). The basic structure of a process declaration is the following:

```
[name:] process [(sensitivity_list)]
   [type_declarations]
   [constant_declarations]
   [variable_declarations]
   [subprogram_declarations]
begin
   sequential_statements
end process [name];
```

The process declaration is contained between the keywords `process` and `end process`. A process may be assigned an optional name for simpler identification of the process in the source code. The name is an identifier and must be followed by the ':' character. This name is also useful for simulation, for example, to set a breakpoint in the simulation execution. The name may be repeated at the end of the declaration, after the keywords `end process`.

The optional sensitivity list is the list of signals to which the process is sensitive. Any event on any of the signals specified in the sensitivity list causes the sequential instructions in the process to be executed, similar to the instructions in a usual program. As opposed to a programming language, in VHDL the `end process` clause does not specify the end of process execution. The process will be executed in an infinite loop. When a

sensitivity list is specified, the process will only suspend after the last statement, until a new event is produced on the signals in the sensitivity list. Note that an event only occurs when a signal changes value. Therefore, the assignment of the same value to a signal does not represent an event.

When the sensitivity list is missing, the process will be run continuously. In this case, the process must contain a `wait` statement to suspend the process and to activate it when an event occurs or a condition becomes true. When the sensitivity list is present, the process cannot contain `wait` statements. The `wait` statement is presented in Section 6.1.1.3.

The declarative part of the process is contained between the `process` and `begin` keywords. This part may contain declarations of types, constants, variables, and subprograms (procedures and functions) that are local to the process. Thus, the declared items can only be used inside the process.

Note

- In a process it is not allowed to declare signals; only constants and variables may be declared.

The statement part of the process starts after the `begin` keyword. This part contains the statements that will be executed on each activation of the process. It is not allowed to use concurrent instructions inside a process.

Example 6.1 presents the declaration of a simple process composed of a single sequential signal assignment.

Example 6.1

```
proc1: process (a, b, c)
begin
   x <= a and b and c;
end process proc1;
```

6.1.1.2. Processes with Incomplete Sensitivity Lists

Some synthesis tools may not check for sensitivity lists of processes. These tools may assume that all signals on the right-hand side of sequential signal assignments are in the sensitivity list. Thus these synthesis tools will interpret the two processes in Example 6.2 to be identical.

Example 6.2

```
proc1: process (a, b, c)
begin
   x <= a and b and c;
end process proc1;

proc2: process (a, b)
begin
   x <= a and b and c;
end process proc2;
```

All synthesis tools will interpret process `proc1` as a 3-input AND gate. Some synthesis tools will also interpret process `proc2` as a 3-input AND gate, even though when this code is simulated, it will not behave as such. While simulating, a change in value of signal `a` or `b` will cause the process to execute, and the value of logical AND of signals `a`, `b`, and `c` will be assigned to signal `x`. However, if signal `c` changes value, the process is not executed, and signal `x` is not updated.

Because it is not clear how a synthesis tool should generate a circuit for which a transition of signal `c` does not cause a change of signal `x`, but for which a change of signals `a` or `b` causes signal `x` to be updated with the logical AND of signals `a`, `b`, and `c`, there are the following alternatives for the synthesis tools:

- Interpret process `proc2` identical to `proc1` (with a sensitivity list that includes all signals on the right-hand side of any signal assignment statement within the process);
- Issue a compile error stating that the process cannot be synthesized without a complete sensitivity list.

The second variant is preferable, because the designer will have to modify the source code so that the functionality of the generated circuit will match the functional simulation of the source code.

Although it is syntactically legal to declare processes without a sensitivity list or a `wait` statement, such processes never suspend. Therefore, if such a process would be simulated, the simulation time would never advance because the initialization phase, in which all processes are executed until suspended, would never complete.

6.1.1.3. `Wait` Statement

Instead of a sensitivity list, a process may contain a `wait` statement. The use of a `wait` statement has two reasons:

- To suspend the execution of a process;
- To specify a condition that will determine the activation of the suspended process.

When a `wait` statement is encountered, the process in which appears that statement suspends. When the condition specified in the `wait` statement is met, the process resumes and its statements are executed until another `wait` statement is encountered. The VHDL language allows several `wait` statements in a process. When used to model combinational logic for synthesis, a process may contain only one `wait` statement.

If a process contains a `wait` statement, it cannot contain a sensitivity list. The process in Example 6.3, which contains an explicit `wait` statement, is equivalent to the process in Example 6.1, which contains a sensitivity list. Both processes will execute when a change of signals `a`, `b`, or `c` occurs.

Example 6.3

```
proc3: process
begin
   x <= a and b and c;
   wait on a, b, c;
end process proc3;
```

`Wait` Statement Forms

There are three forms of the `wait` statement:

```
wait on sensitivity_list;
wait until conditional_expresion;
wait for time_expression;
```

The `wait on` statement has been illustrated in the preceding examples. The `wait until` statement suspends a process until the specified condition becomes true due to a change of any signal listed in the conditional expression. Note that if none of the signals in that expression changes, the process will not be activated, even if the conditional expression is true. The following examples present several forms of the `wait until` statement:

```
wait until signal = value;
wait until signal'event and signal = value;
wait until not signal'stable and signal = value;
```

where *`signal`* is the name of a signal, and *`value`* is the value tested. If the signal is of type `bit`, then if the value tested is '1', the statement will wait for the rising edge of the signal, and if it is '0', the statement will wait for the falling edge of the signal.

The `wait until` statement may be used to implement a synchronous operation. Usually, the signal tested is a clock signal. For example, waiting for the rising edge of the clock signal may be expressed in the following ways:

```
wait until clk = '1';
wait until clk'event and clk = '1';
wait until not clk'stable and clk = '1';
```

For descriptions that are to be synthesized, the `wait until` statement must be the first statement in the process. Because of this, synchronous logic described by a `wait until` statement cannot be asynchronously reset.

The `wait for` statement allows to suspend the execution of a process for a specified time, for example:

```
wait for 10 ns;
```

It is possible to combine several conditions of the `wait` statement in a united condition. In Example 6.4, the process `proc4` will be activated when one of the signals `a` or `b` changes, but only when the value of the `clk` signal is '1'.

Example 6.4

```
proc4: process
begin
   wait on a, b until clk = '1';
   ...
end process proc4;
```

`Wait` Statement Positioning

Usually, the `wait` statement appears either at the beginning of a process, or at the end of it. In Example 6.3, the `wait` statement appears at the end of the process. It was mentioned that this form is equivalent to the form that contains a sensitivity list. This equivalence is due to the way in which the simulation of a model is performed. At the initialization phase of the simulation, all the processes in the model are run once. If the process contains a sensitivity list, all the statements of the process will be run once. In a process that contains a `wait` statement and it appears at the end of the process, at the initialization phase all the statements before the `wait` statement are run once, as in a process that contains a sensitivity list.

When the `wait` statement appears at the beginning of the process, the simulation will be performed differently, since at the initialization phase the process will be suspended without executing any statement of it. Thus, a process with a `wait` statement placed at the beginning of the process is not equivalent to a process that contains a sensitivity list. Since the initialization phase of the simulation has no hardware equivalent, there will be no differences when synthesizing the two processes, with the `wait` statement placed at the end or at the beginning, respectively.

However, because of the difference between simulation and synthesis as regards the initialization phase of the simulation, there may be differences between the behavior of the model at simulation and the operation of the circuit generated by synthesis. It is possible that the model working correctly at simulation to be synthesized into a circuit whose operation is incorrect.

6.1.1.4. Combinational and Sequential Processes

Both combinational and sequential processes are interpreted in the same way, the only difference being that for sequential processes the output signals are stored into registers. A simple combinational process is described in Example 6.5.

Example 6.5

```
proc5: process
begin
   wait on a, b;
   z <= a and b;
end process proc5;
```

This process will be implemented by synthesis as a two-input AND gate.

For a process to model combinational logic, it must contain in the sensitivity list all the signals that are inputs of the process. In other words, the process must be reevaluated every time one of the inputs to the circuit it models changes. In this way combinational logic is correctly modeled.

If a process is not sensitive to all its inputs and it is not a sequential process, then it cannot be synthesized, since there is no hardware equivalent of such a process. Not all synthesis tools enforce such a rule, so great care should be taken in the design of combinational processes in order not to introduce errors in the design. Such errors will cause subtle differences between the simulated model and the circuit obtained by synthesis, because a non-combinational process is interpreted by the synthesizer as a combinational circuit.

If a process contains a `wait` statement or an `if` *`signal`*`'event` statement, the process will be interpreted as a sequential process. Hence the process in Example 6.6 will be interpreted as a sequential process.

Example 6.6

```
proc6: process
begin
   wait until clk = '1';
   z <= a and b;
end process proc6;
```

By synthesizing this process, the circuit in Figure 6.1 will result, where a flip-flop is added on the output.

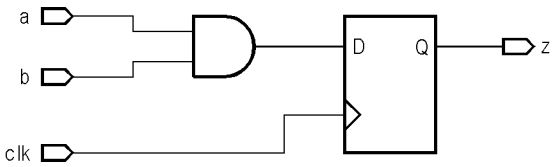

Figure 6.1. Synthesis result of a sequential process.

6.1.2. Sequential Signal Assignment Statement

Signals represent the interface between the concurrent domain of a VHDL model and the sequential domain within a process. A VHDL model is composed of several processes that communicate via signals. At simulation, all hierarchy of the model is removed and only the processes and signals remain. The simulator alternates between updating signal values and then running processes activated by changes of the signals listed in their sensitivity lists.

6.1.2.1. Sequential Assignment Statement Execution

Signal assignment may be performed using a sequential statement or a concurrent statement. The sequential statement may only appear inside a process, while the concurrent statement may only appear outside processes. The sequential signal assignment has a single form, the simple one, which is an unconditional assignment. The concurrent signal assignment has, in addition to its simple form (presented in Section 6.2.3.1), two other forms: the conditional assignment (Section 6.2.3.2) and the selective assignment (Section 6.2.3.3). The sequential signal assignment has the same syntax as the simple form of the concurrent signal assignment; the difference between them results from context.

The sequential signal assignment has the following syntax:

signal <= *expression* [**after** *delay*];

As a result of executing this statement in a process, the expression on the right-hand side of the assignment symbol is evaluated and an event is scheduled to change the value of the signal. The simulator will only change the value of a signal when the process suspends, and, if the `after` clause is used, after the delay specified from the current time. Therefore, in a process the signals will be updated only after executing all the statements of the process or when a `wait` statement is encountered.

Typically, synthesis tools do not allow to use `after` clauses, or they ignore these clauses. The `after` clauses are ignored not only because their interpretation for synthesis is not specified by standards, but also because it would be difficult to guarantee the results of such delays. For example, it is not clear whether the delay should be interpreted as a minimum or maximum propagation delay. Also, it is not clear how the synthesis tool should proceed if a delay specified in the source code cannot be assured.

A consequence of the way in which signal assignments within processes are executed is that when more than one value is assigned to the same signal, only the last assignment will be effective. Thus, the two processes in Example 6.7 are equivalent.

Example 6.7

```
proc7: process (a)
begin
   z <= '0';
   z <= a;
end process proc7;
```

```
proc8: process (a)
begin
   z <= a;
end process proc8;
```

In conclusion, the following important aspects should be taken into consideration when signal assignment statements are used inside processes:

- Any signal assignment becomes effective only when the process suspends. Until that moment, all signals keep their old values.
- Only the last assignment to a signal will be effectively executed. Therefore, it would make no sense to assign more than one value to a signal in the same process.

6.1.2.2. Feedback

Another consequence of the way in which signal assignments are executed is that any reading of a signal that is also assigned to in the same process will return the value assigned to the signal in the previous execution of the process. Reading a signal and assigning a value to it in the same process is equivalent to a *feedback*. In a combinational process, the previous value is an output of the combinational logic and so the feedback is asynchronous. In a sequential process, the previous value is the value stored in a latch or register, so the feedback is synchronous.

Example 6.8 presents a process in which a synchronous feedback is used. The process describes a 4-bit counter. The signal `count` is of type `unsigned`, type defined in the `numeric_bit` package. Observe that the signal `count` is declared outside the process.

Example 6.8

```
library ieee;
use ieee.numeric_bit.all;

signal count: unsigned (3 downto 0);
process
begin
   wait until clk = '1';
   if reset = '1' then
      count <= "0000";
   else
      count <= count + "0001";
   end if;
end process;
```

The circuit that results from synthesizing the process in the previous example is shown in Figure 6.2.

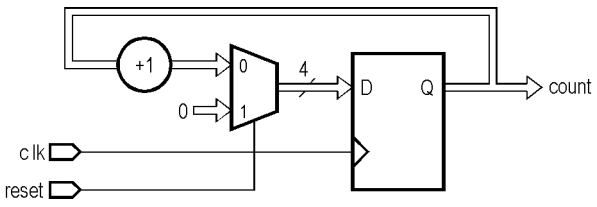

Figure 6.2. Example of synchronous feedback.

6.1.2.3. Inertial Delay

In the VHDL language there are two types of delays that can be used to model real systems. These are the inertial delay and the transport delay. Note that these delays cannot be used for logic synthesis.

The inertial delay is the default delay and it is used when the type of delay is not specified. The `after` clause assumes by default the inertial delay. In a model with inertial delay, two consecutive changes of an input signal value will not change an output signal value if the time between these changes is shorter than the specified

delay. This delay represents the inertia of the real circuit. If, for example, certain pulses of short periods of the input signals occur, the output signals will remain unchanged.

Figure 6.3 illustrates the inertial delay with a simple buffer. The buffer with a delay of 20 ns has an input *A* and an output *B*. Signal *A* changes from '0' to '1' at 10 ns and from '1' to '0' at 20 ns. The input signal pulse has a duration of 10 ns, which is shorter than the delay introduced by the buffer. As a result, the output signal *B* remains '0'.

The buffer in Figure 6.3 can be modeled by the following assignment statement:

```
b <= a after 20 ns;
```

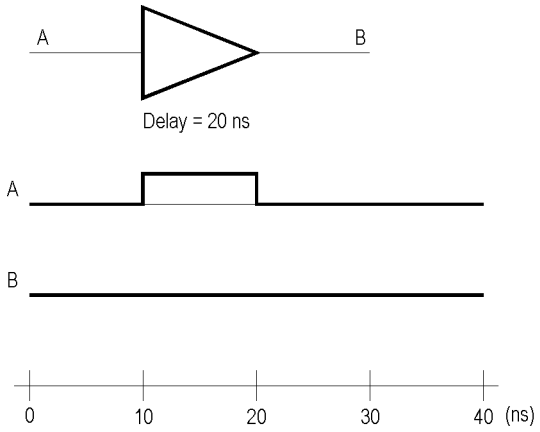

Figure 6.3. Illustration of the inertial delay.

6.1.2.4. Transport Delay

The transport delay must be indicated explicitly by the `transport` keyword. This represents the delay of an interconnection, in which the effect of a pulse in an input signal is propagated to the output with the specified delay, regardless of the duration of that pulse. The transport delay is especially useful for modeling transmission lines and interconnections between components.

Considering the same buffer in Figure 6.3 and the input signal *A* with the same waveform, if the inertial delay is replaced with the transport delay, the output signal *B* will have the form shown in Figure 6.4. The pulse on the input signal is propagated unchanged to the output with a delay of 20 ns.

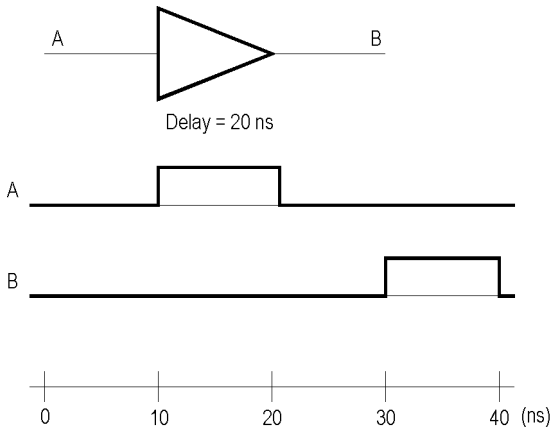

Figure 6.4. Illustration of the transport delay.

If the transport delay is used, the buffer in Figure 6.4 may be modeled by the following assignment statement:

```
b <= transport a after 20 ns;
```

6.1.3. Variables

Restrictions enforced on signals reduce the possibilities to use them. Because signals can only hold the last value assigned to them, they cannot be used to store intermediary results within a process. Another drawback is that the new values are not assigned to signals when the assignment statement executes, but only after the process execution suspends. This causes difficulties in analyzing descriptions.

As opposed to signals, variables can be declared inside processes and can be used to store intermediary results. However, variables can be used only in the sequential domain of the VHDL language, that is, inside processes and subprograms, and cannot be declared or used directly in an architecture. Thus, they are local to that process or subprogram.

6.1.3.1. Declaring and Initializing Variables

Like signals, variables must be declared before they are used. A variable declaration is similar to that of a signal, but the `variable` keyword is used. This declaration specifies the type of the variable. Optionally, for scalar variables a constrained range may be specified, and for array variables a constrained index may be specified. For both types of variables, an initial value may be specified. Example 6.9 illustrates variable declarations and initializations.

Example 6.9

```
variable a, b, c: bit;
variable x, y: integer;
variable index range 1 to 10 := 1;
variable cycle_t: time range 10 ns to 50 ns := 10 ns;
variable mem: bit_vector (0 to 15);
```

A variable is given an initial value at the initialization phase of the simulation. This initial value is either that specified explicitly in the variable declaration, or a default value, which is the left value of the type. For example, for type `bit` the initial value is '0', and for type `integer` this value is –2,147,483,647.

For synthesis, there is no hardware interpretation of initial values, so synthesis tools ignore initial values or signal errors.

6.1.3.2. Variable Assignment Statement

A variable assignment statement replaces the current value of a variable with a new value specified by an expression. The expression may contain variables, signals, and literals. The variable and the result of expression evaluation must be of the same type. The syntax of the variable assignment statement is the following:

variable := *expression*;

This statement is similar to the assignments in most of the programming languages. As opposed to a sequential signal assignment, a variable assignment is executed immediately, that is, in zero simulation time.

The following main differences exist between signal assignment and variable assignment:

- For a signal assignment an event is scheduled to update the signal value, and this update is executed only when the process execution suspends. For a variable assignment, an event is not scheduled and the update is executed instantly.
- For a signal assignment a delay may be specified, while a variable assignment cannot be delayed.
- In a process only the last assignment to a signal is effective. Instead, numerous assignments to a variable may exist in a process, and all are effective.

Example 6.10 illustrates the use of variables to store intermediary results.

Example 6.10

```
entity add_1 is
   port (a, b, cin: in bit;
         s, cout: out bit);
end add_1;
```

```
architecture functional of add_1 is
begin
   process (a, b, cin)
      variable s1, s2, c1, c2: bit;
   begin
      s1 := a xor b;
      c1 := a and b;
      s2 := s1 xor cin;
      c2 := s1 and cin;
      s <= s2;
      cout <= c1 or c2;
   end process;
end functional;
```

The preceding example describes a full adder. This is not the optimal way to describe a full adder, but illustrates the use of variables. The result is generated by creating two half-adders; the first generates the outputs `s1` and `c1`, and the second the outputs `s2` and `c2`. Finally, the outputs are assigned to the output ports `s` and `cout` by signal assignment statements, since ports are signals.

6.1.4. `If` Statement

6.1.4.1. Syntax and Execution of an `if` Statement

The `if` statement selects one or more statement sequences for execution, based on the evaluation of a condition corresponding to that sequence. The syntax of this statement is the following:

```
if condition then statement_sequence
   [elsif condition then statement_sequence...]
   [else statement_sequence]
end if;
```

Each condition is a Boolean expression, which is evaluated to `TRUE` or `FALSE`. More than one `elsif` clauses may be present, but a single `else` clause may exist. First, the condition after the `if` keyword is evaluated, and if it evaluates true, the corresponding statement or statements are executed. If it evaluates false and the `elsif` clause is present, the condition after this clause is evaluated; if this condition evaluates true, the corresponding statement or statements are executed. Otherwise, if there are other `elsif` clauses, the evaluation of their conditions continues. If none of the conditions evaluated is true, the sequence of statements corresponding to the `else` clause is executed, if this clause is present.

Example 6.11

```
process (a, b)
begin
   if a = b then
      result <= 0;
   elsif a < b then
      result <= -1;
   else
      result <= 1;
   end if;
end process;
```

6.1.4.2. Synthesis Interpretation of an `if` Statement

An `if` statement may be implemented by a multiplexer. First consider the `if` statement without any `elsif` branch. Example 6.12 presents the use of such a statement to describe a comparator.

Example 6.12

```
library ieee;
use ieee.numeric_bit.all;
entity comp is
   port (a, b: in unsigned (7 downto 0);
```

```
        equal: out bit);
end comp;

architecture functional of comp is
begin
   process (a, b)
   begin
      if a = b then
         equal <= '1';
      else
         equal <= '0';
      end if;
   end process;
end functional;
```

The previous example tests the equality of two signals of type `unsigned`, representing two 8-bit unsigned integers, and gives a result of type `bit`. The resulting circuit is shown in Figure 6.5. Note that, as with other examples, in practice the synthesis tool will eliminate the inefficiencies in the circuit (in this case, the constant inputs to the multiplexer) to give a minimal solution.

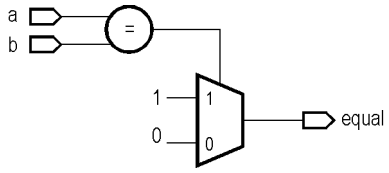

Figure 6.5. Synthesis of an `if` statement.

A multi-branch `if` statement, in which at least one `elsif` clause appears, is implemented by a multistage multiplexer. Consider the `if` statement in Example 6.13.

Example 6.13

```
process (a, b, c, s0, s1)
begin
   if s0 = '1' then
      z <= a;
   elsif s1 = '1' then
      z <= b;
   else
      z <= c;
   end if;
end process;
```

The result of implementing the `if` statement from the previous example is presented in Figure 6.6. This circuit is equivalent to that resulted by implementing a conditional signal assignment (presented in Section 6.2.3.2).

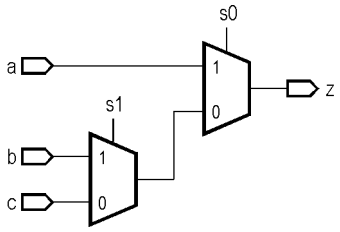

Figure 6.6. Synthesis of a multi-branch `if` statement.

The conditions in the successive branches of an `if` statement are evaluated independently. In the previous example, the conditions involve the two signals `s0` and `s1`. There can be any number of conditions, and each of them is independent of the others. The structure of the `if` statement ensures that the earlier conditions are

tested first. In this example, signal `s0` has been tested before signal `s1`. This priority is reflected in the generated circuit, where the multiplexer controlled by signal `s0` is nearer to the output than the multiplexer controlled by signal `s1`.

It is important to remember the existence of this priority for condition testing, so that redundant tests can be eliminated. Consider the `if` statement in Example 6.14, which is equivalent to the `if` statement in Example 6.13.

Example 6.14

```
process (a, b, c, s0, s1)
begin
   if s0 = '1' then
      z <= a;
   elsif s0 = '0' and s1 = '1' then
      z <= b;
   else
      z <= c;
   end if;
end process;
```

The additional condition `s0 = '0'` is redundant since it will be tested only if the first condition of the `if` statement is false. It is recommended to avoid such redundancies, because there is no guarantee that they will be detected and eliminated by the synthesis tool.

For multi-branch `if` statements, normally each condition will be dependent on different signals and variables. If every branch is dependent on the same signal, then it is more advantageous to use a `case` statement. The `case` statement is presented in Section 6.1.5.

6.1.4.3. Incomplete `if` Statements

In the examples presented so far, all the `if` statements have been complete. In other words, the target signal has been assigned a value under all possible conditions. However, there are two situations when a signal does not receive a value: when the `else` clause of the `if` statement is missing, and when the signal is not assigned to a value in some branches of the `if` statement. In both cases the interpretation is the same. In the situations when a signal does not receive a value, its previous value is preserved.

The problem is what the previous value of the signal is. If there is a previous assignment statement in which the signal appears as target, then the previous value comes from that assignment statement. If not, the value comes from the previous execution of the process, leading to feedback in the circuit.

The first case is illustrated in Example 6.15.

Example 6.15

```
process (a, b, enable)
begin
   z <= a;
   if enable = '1' then
      z <= b;
   end if;
end process;
```

In this case, the `if` statement is incomplete because the `else` clause is missing. In the `if` statement, the signal `z` gets a value if the condition `enable = '1'` is true, but remains unassigned if the condition is false. The previous value comes from the unconditional assignment before the `if` statement.

The `if` statement of Example 6.15 is equivalent to the `if` statement of Example 6.16.

Example 6.16

```
process (a, b, enable)
begin
   if enable = '1' then
      z <= b;
   else
      z <= a;
```

```
   end if;
end process;
```

When the `if` statement is incomplete and there is no previous assignment, then a feedback will exist from the output of the circuit to the input. This is because the value of the signal from the previous execution of the process is preserved and it becomes the value in the current execution of the process.

This form of the `if` statement is used to describe a flip-flop or a register with an enable input, as in Example 6.17.

Example 6.17

```
process
begin
   wait until clk = '1';
   if en = '1' then
      q <= d;
   end if;
end process;
```

The signal `q` is updated with the new value of signal `d` when the condition is true, but is not updated when the condition is false. In this case, the previous value of signal `q` is preserved by sequential feedback of `q`. The resulting circuit is presented in Figure 6.7.

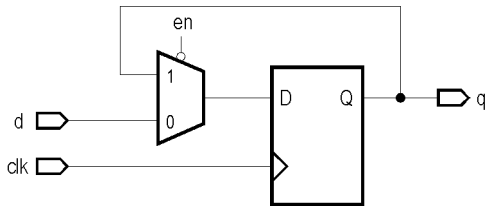

Figure 6.7. Synthesis of an incomplete `if` statement.

The `if` statement of Example 6.17 is equivalent to the complete `if` statement of Example 6.18.

Example 6.18

```
process
begin
   wait until clk = '1';
   if en = '1' then
      q <= d;
   else
      q <= q;
   end if;
end process;
```

When the condition is false, the signal `q` is assigned to itself, which is equivalent to preserving its previous value.

One of the most common errors encountered in VHDL descriptions targeted for synthesis is the unintended introduction of feedback in the circuit due to an incomplete `if` statement. In order to avoid the introduction of feedback, the designer must ensure that every signal assigned to in an `if` statement within a process (which is therefore an output signal of the process) receives a value under every possible combination of conditions. In practice, there are two possibilities of doing this: to assign a value to output signals in every branch of the `if` statement and including the `else` clause, or to initialize signals with an unconditional assignment before the `if` statement.

In the following example, although the `if` statement looks complete, different signals are being assigned a value in each branch of the `if` statement. Thus both signals `z` and `y` will have asynchronous feedback.

Example 6.19

```
process
begin
```

```
   wait on a, b, c;
   if c = '1' then
      z <= a;
   else
      y <= b;
   end if;
end process;
```

Another example is where there is a redundant test for a condition which must be true (Example 6.20).

Example 6.20

```
process
begin
   wait on a, b, c;
   if c = '1' then
      z <= a;
   elsif c = '0' then
      z <= b;
   end if;
end process;
```

In this case, although the `if` statement looks complete (assuming that signal `c` is of type `bit`), each of the conditions in the `if` statement is synthesized independently. The synthesis tool will therefore not detect that this second condition is redundant. Thus the `if` statement is synthesized as a three-way multiplexer, the third input being the missing `else` condition which is the feedback of the previous value. The circuit synthesized for this example is shown in Figure 6.8.

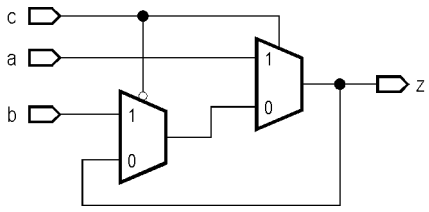

Figure 6.8. Synthesis of an `if` statement with a redundant condition.

6.1.4.4. `If` Statements with Variables

So far, in the `if` statements only signals were used. The same rules apply when using variables, with a single difference. Like a signal, if a variable is assigned to only in some branches of the `if` statement, then the previous value is preserved by feedback. Unlike the case when a signal is used, the reading and writing of a variable in the same process will result in feedback only if the read occurs before the write. In this case, the value read is the previous value of the variable. In the case when a signal is used, a read and a write in the same process will always result in feedback.

This observation may be used to create registers or counters using variables. Remember that a sequential process is interpreted by synthesis by placing a flip-flop or register on every signal assigned to in the process. This means that normally variables are not written to flip-flops or registers. However, if there is feedback of a previous variable value, then this feedback is implemented via a flip-flop or register to make the process synchronous.

Example 6.21 describes a counter using the `unsigned` integer type. When a value of type `unsigned` is incremented, if the value is the highest value of the range, then the lowest value of the range is obtained.

Example 6.21

```
process
   variable count: unsigned (7 downto 0);
begin
   wait until clk = '1';
   if reset = '1' then
      count := "00000000";
```

```
   else
      count := count + 1;
   end if;
   result <= count;
end process;
```

In this example, in the `else` branch of the `if` statement the previous value of the `count` variable is being read to calculate the next value. This results in a feedback.

Note that in this example actually two registers are created. According to the feedback rules, variable `count` will be registered. Signal `result` will also be registered, because all signals assigned to in a sequential process will be registered. This extra register will always contain the same value as the register for variable `count`. The synthesis tool will normally eliminate this redundant register.

6.1.5. Case Statement

Like the `if` statement, the `case` statement selects for execution one of several alternative statement sequences based on the value of an expression. Unlike the `if` statement, the expression does not need to be Boolean, but it may be represented by a signal, variable, or expression of any discrete type (an enumeration or an integer type) or a character array type (including `bit_vector` and `std_logic_vector`). The `case` statement is used when there are a large number of possible alternatives. This statement is more readable than an `if` statement with many branches, allowing to easily identify a value and the associated statement sequence.

The syntax of the `case` statement is the following:

```
case expression is
   when options_1 =>
      statement_sequence
   ...
   when options_n =>
      statement_sequence
   [when others =>
      statement_sequence]
end case;
```

The `case` statement contains several `when` clauses, and each of these specifies one or more options. The options represent either an individual value or a set of values to which the expression of the `case` statement is compared. If the expression is equal to the individual value or one of the values from the set, the statement sequence specified after the `=>` symbol is executed. A statement sequence may also be represented by the `null` statement. As opposed to some programming languages, the statements of a sequence do not need to be included between the `begin` and `end` keywords. The `others` clause may be used to specify the execution of a statement sequence when the expression value is not equal to none of the values specified by the `when` clauses.

If an option is represented by a set of values, either the individual values of the set may be specified, separated by the "|" symbol (meaning "or"), or the range of the values, or a combination of these, as shown in the following example:

```
case expression is
   when val =>
      statement_sequence
   when val1 | val2 | ... | valn =>
      statement_sequence
   when val3 to val4 =>
      statement_sequence
   when val5 to val6 | val7 to val8 =>
      statement_sequence
   ...
   when others =>
      statement_sequence
end case;
```

Notes

- In a `case` statement all the possible values of the selection expression must be enumerated, based on the expression's type or subtype. For instance, if the selection expression is a `std_logic_vector` of 2

bits, 81 values must be enumerated, because for a single bit 9 possible values can exist. If all the possible values are not enumerated, the `others` clause must be used.

- The `others` clause must be the last of all options.

Example 6.22 presents a process for sequencing through the values of an enumeration type representing the states of a traffic light. The process is described with a `case` statement.

Example 6.22

```
type type_color is (red, yellow, green);
process (color)
   case color is
      when red =>
         next_color <= green;
      when yellow =>
         next_color <= red;
      when green =>
         next_color <= yellow;
   end case;
end process;
```

Example 6.23 defines an entity and an architecture for a 2-input EXCLUSIVE OR gate. The gate is described behaviorally with a process containing a `case` statement.

Example 6.23

```
library ieee;
use ieee.std_logic_1164.all;
entity xor2 is
   port (a, b: in std_logic;
         x: out std_logic);
end xor2;

architecture arch_xor2 of xor2 is
begin
   process (a, b)
      variable temp: std_logic_vector (1 downto 0);
   begin
      temp := a & b;
      case temp is
         when "00"   => x <= '0';
         when "01"   => x <= '1';
         when "10"   => x <= '1';
         when "11"   => x <= '0';
         when others => x <= '0';
      end case;
   end process;
end arch_xor2;
```

A `case` statement is implemented by a multiplexer, just like the `if` statement. The difference is that all the choices depend an the same input and are mutually exclusive. This allows some optimizations of the control conditions compared with a multi-branch `if` statement where no dependencies between the conditions will be assumed.

6.1.6. Loop Statements

Loop statements allow the repeated execution of a statement sequence, for example, processing each element of an array in the same way. In the VHDL language there are three types of loop statements: the simple `loop`, the `while loop`, and the `for loop`. The simple `loop` statement specifies an indefinite repetition of some statements. With the `while loop` the statements that form the loop body are repeated until a condition specified becomes false, and with the `for loop` the statements of the loop body are repeated a number of times specified by a counter.

Note

- The only loop statement that can be used for logic synthesis is the `for loop` statement, because in this case the number of iterations is fixed.

6.1.6.1. `loop` Statement

The simple `loop` statement has the following syntax:

```
[label:] loop
   statement_sequence
end loop [label];
```

The statement has an optional label that may be used to identify the statement. The `loop` statement has the effect of repeating the statements within the statement body an unlimited number of times. With this statement, the only possibility to end the execution is to use an `exit` statement (presented in Section 6.1.6.5).

6.1.6.2. `while loop` Statement

The `while loop` statement is a conditional loop statement. The syntax of this statement is the following:

```
[label:] while condition loop
   statement_sequence
end loop [label];
```

The condition is tested before each execution of the loop. If the condition is true, the sequence of statements within the loop body is executed, after which the control is transferred to the beginning of the loop. The loop execution terminates when the condition tested becomes false, in which case the statement that follows the `end loop` clause is executed.

Example 6.24

```
process
   variable count: integer := 0;
begin
   wait until clk = '1';
   while level = '1' loop
      count := count + 1;
      wait until clk = '0';
   end loop;
end process;
```

The process in Example 6.24 counts the rising edges of the `clk` clock signal while the `level` signal is '1'. The stability of the `level` signal is not tested.

The loop statements may be nested on several levels. Thus, the loop body of a `while loop` statement may contain another loop statement, in particular a `while loop` statement, as illustrated in the following example.

```
E1:    while i < 10 loop
E2:       while j < 20 loop
          ...
          end loop E2;
       end loop E1;
```

6.1.6.3. `for loop` Statement

For the repeated execution of a sequence of statements a fixed number of times, the `for loop` statement may be used. The syntax of this statement is the following:

```
[label:] for counter in range loop
   statement_sequence
end loop [label];
```

In a `for loop` statement an iteration counter and a range are specified. The statements within the loop body are executed while the counter is in the specified range. After an iteration, the counter is assigned the next value from the range. The range may be an ascending one, specified by the `to` keyword, or a descending one, specified by the `downto` keyword. This range may also be specified as an enumeration type or subtype, when the range boundaries are not explicitly specified in the `for loop` statement. The range boundaries are determined by the compiler from the type or subtype declaration.

The `for loop` statement in Example 6.25 computes the squares of the integer values between 1 and 10, and stores them into the `i_square` array.

Example 6.25

```
for i in 1 to 10 loop
   i_square (i) <= i * i;
end loop;
```

In this example, the iteration count is of type `integer` by default, since its type has not been defined explicitly. The complete form of the domain declaration for the iteration count is similar to that of a type. For the previous example, the `for` clause can also be written in the following form:

```
for i in integer range 1 to 10 loop
```

In some programming languages, within a loop a value may be assigned to the iteration count (in the previous example, `i`). The VHDL language, however, does not allow to assign a value to the iteration count or to use it as an input or output parameter of a procedure. The counter may be used in an expression, provided that its value is not modified. Another aspect related to the iteration count is that there is no need to declare it explicitly within the process. This counter is declared implicitly as a local variable of the `loop` statement by specifying it after the `for` keyword. If there is another variable with the same name within the process, they will be treated as separate variables.

The synthesis interpretation of the `for loop` statement is that a new copy is generated for the circuit described by the statement in each iteration of the loop. The use of the `for loop` statement to generate a circuit is illustrated in Example 6.26.

Example 6.26

```
entity match_bits is
   port (a, b: in bit_vector (7 downto 0);
         matches: out bit_vector (7 downto 0));
end match_bits;

architecture functional of match_bits is
begin
   process (a, b)
   begin
      for i in 7 downto 0 loop
         matches (i) <= not (a(i) xor b(i));
      end loop;
   end process;
end functional;
```

The process in the previous example generates a set of 1-bit comparators to compare the bits of the same order of vectors `a` and `b`. The result is stored into the `matches` vector, which will contain '1' wherever the bits of the two vectors match and '0' otherwise.

The process in the previous example is equivalent to the following process:

```
process (a, b)
begin
   matches (7) <= not (a(7) xor b(7));
   matches (6) <= not (a(6) xor b(6));
   matches (5) <= not (a(5) xor b(5));
   matches (4) <= not (a(4) xor b(4));
   matches (3) <= not (a(3) xor b(3));
   matches (2) <= not (a(2) xor b(2));
   matches (1) <= not (a(1) xor b(1));
   matches (0) <= not (a(0) xor b(0));
end process;
```

In this case, the ordering of the iteration count is not relevant, since there is no connection between the replicated blocks of the circuit. This ordering becomes important where there is a connection from one replicated block to another. This connection is usually created by a variable that stores a value in one iteration of the loop, value that is read in another iteration of the loop. It is usually necessary to initialize such a variable prior to entering the loop. Example 6.27 presents such a circuit.

Example 6.27

```
library ieee;
use ieee.numeric_bit.all;
entity count_ones is
   port (v: in bit_vector (15 downto 0);
         count: out signed (3 downto 0));
end count_ones;

architecture functional of count_ones is
begin
   process (v)
      variable result: signed (3 downto 0);
   begin
      result := (others => '0');
      for i in 15 downto 0 loop
         if v(i) = '1' then
            result := result + 1;
         end if;
      end loop;
      count <= result;
   end process;
end functional;
```

This example is a combinational circuit which counts the number of bits in vector `v` that are set to '1'. The result is accumulated during the execution of the process in a variable called `result` and then assigned to the output signal `count` at the end of the process. The loop body – an `if` statement containing a variable assignment – represents a block containing a multiplexer and an adder, which will be generated by synthesis for each iteration of the loop. The output of a block becomes the input `result` of the next block. Figure 6.9 presents the circuit generated for this loop.

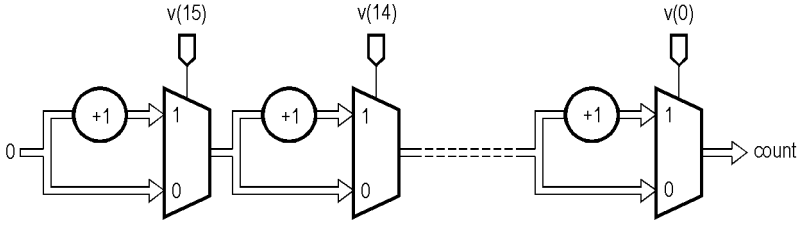

Figure 6.9. Synthesis of a `for loop` statement.

In Example 6.27, the range of the `for loop` statement's iteration count was specified explicitly as `15 downto 0`. In practice, this explicit form of the range is very rarely used when accessing arrays; it is recommended to use array attributes to specify the loop bounds. There are four possibilities, depending on whether the array being accessed has an ascending or a descending range and the order in which the array elements should be visited.

1. If the array elements should be visited from left to right regardless of whether the array has an ascending or descending range, the `'range` attribute is used:

   ```
   for i in v'range loop
   ```

2. If the array elements should be visited from right to left, then the `'reverse_range` attribute is used:

   ```
   for i in v'reverse_range loop
   ```

3. If the array elements should be visited from the lowest index to the highest index regardless of whether the array has an ascending or descending range, the `'low` and `'high` attributes are used:

```
for i in v'low to v'high loop
```

4. Finally, if the array elements should be visited from the highest index to the lowest index, the `'high` and `'low` attributes are used with the `downto` keyword:

```
for i in v'high downto v'low loop
```

When writing subprograms, where it is not known in advance if an array variable or signal will have an ascending range or a descending range, it becomes particularly important to choose the correct loop range. This problem is presented in more details in Laboratory No. 9. For array types used to represent integer values, such as the types `signed` and `unsigned` defined in the `numeric_bit` and `numeric_std` packages, the convention is that the leftmost bit is the most significant bit and the rightmost bit is the least significant bit, regardless the array range. This means that the correct way to access such an array is to use the `'range` attribute or the `'reverse_range` attribute.

Thus, to access an array `n` representing an integer from the most significant bit to the least significant bit, the `'range` attribute may be used:

```
for i in n'range loop
```

To access the same array from the least significant bit to the most significant bit, the `'reverse_range` attribute may be used:

```
for i in n'reverse_range loop
```

Because of the synthesis interpretation of the `for loop` statement, the bounds of the loop must be constant. This means that the bounds cannot be defined using the value of a variable or signal. This constraint makes some circuits difficult to describe. For example, consider a case when the number of leading zeros of an integer value must be counted, and so the number of iterations is not known in advance. The description of this types of circuits using loop statements is facilitated by the `next` and `exit` statements.

6.1.6.4. `Next` Statement

There are cases when the statements remaining in the current iteration of a loop must be skipped and the execution must be continued with the next iteration. In these cases the `next` statement may be used. The syntax of this statement is the following:

```
next [label] [when condition];
```

When a `next` statement is encountered in a loop body, the execution of the current iteration is skipped and the control is passed to the beginning of the loop statement either unconditionally, if the `when` clause is not present, or conditionally, if this clause is present. The iteration count is updated and, if the bound of the range is not reached, the execution will continue with the first statement of the loop body. Otherwise, the execution of the loop statement will terminate.

When there are several levels of loop statements (a loop contained in another loop), a label may be specified in the `next` statement; this applies to the innermost enclosing loop statement. The label only has the effect of increasing the readability of the description and cannot be different from the label of the current loop statement.

A `next` statement may be used as a substitute of an `if` statement for conditionally executing a group of statements. To implement the `next` statement the same hardware is required as to implement the `if` statement. The designer has to choose whether to use a `next` statement or an `if` statement.

As an example, consider the same circuit to count the number of '1' bits in a vector. Example 6.28 presents the modified description of this circuit. A `next` statement is used instead of the `if` statement, by which one iteration is abandoned when the value of the current element is '0', so that the counter is not incremented.

Example 6.28

```
library ieee;
use ieee.numeric_bit.all;
entity count_ones is
   port (v: in bit_vector (15 downto 0);
         count: out signed (3 downto 0));
end count_ones;

architecture functional of count_ones is
begin
```

```
   process (v)
      variable result: signed (3 downto 0);
   begin
      result := (others => '0');
      for i in v'range loop
         next when v(i) = '0';
         result := result + 1;
      end loop;
      count <= result;
   end process;
end functional;
```

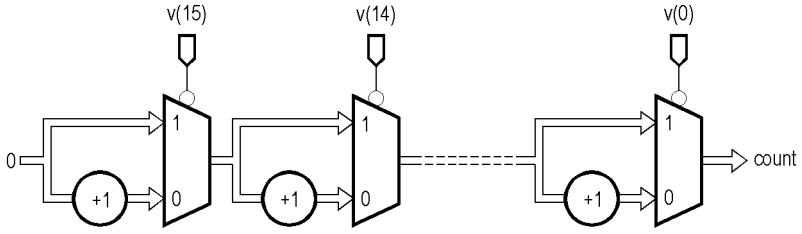

Figure 6.10. Synthesis of a `for loop` statement containing a `next` statement.

Figure 6.10 presents the circuit generated by synthesizing the description from the previous example. The only difference between this circuit and the circuit in Figure 6.9 is that the multiplexers' control logic is inverted. However, the two circuits are functionally equivalent.

6.1.6.5. `Exit` Statement

There are situations when the execution of a loop statement must be stopped completely, either as a result of an error occurred during the execution of a model or due to the fact that the processing must be terminated before the iteration count exceeds its range. In these situations, an `exit` statement may be used. The syntax of this statement is the following:

```
exit [label] [when condition];
```

There are three forms of the `exit` statement. The first is the one without a label and without a condition specified by a `when` clause. In this case, the execution of the current loop will be terminated unconditionally. When the `exit` statement appears in a loop statement placed inside another loop statement, only the execution of the innermost loop statement will be stopped, but the execution of the outer loop statement will continue.

If a label of a loop statement is specified in the `exit` statement, when the `exit` statement is encountered the control is transferred to the specified label.

When the `exit` statement contains a `when` clause, the execution of the loop statement will be stopped only if the condition specified by this clause is true. The next statement executed depends on the presence or absence of a label within the statement. If a label of a loop statement is specified, the next statement executed will be the first statement of the loop specified by that label. If a label is not specified, the next statement executed will be the one after the `end loop` clause of the current loop statement.

The `exit` statement may be used to terminate the execution of a simple `loop` statement, as shown in Example 6.30.

Example 6.30

```
E3:   loop
         a := a + 1;
         exit E3 when a > 10;
      end loop E3;
```

Example 6.31 presents the description of a circuit to count the number of trailing zeros of a vector of bits. Each element of the vector representing an integer value is tested, and if an element is '1', the loop is terminated with the `exit` statement.

Example 6.31

```
library ieee;
use ieee.numeric_bit.all;
entity count_zeros is
   port (v: in bit_vector (15 downto 0);
         count: out signed (3 downto 0));
end count_zeros;

architecture functional of count_zeros is
begin
   process (v)
      variable result: signed (3 downto 0);
   begin
      result := (others => '0');
      for i in v'reverse_range loop
         exit when v(i) = '1';
         result := result + 1;
      end loop;
      count <= result;
   end process;
end functional;
```

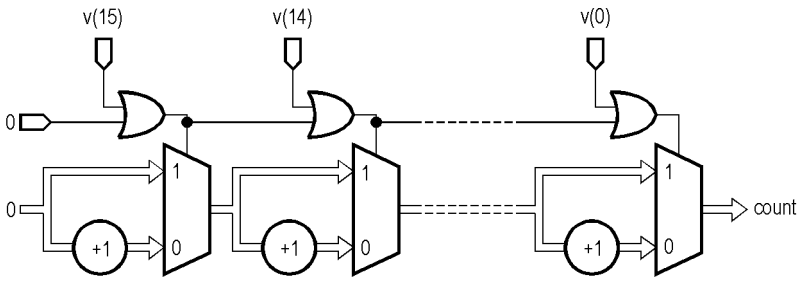

Figure 6.11. Synthesis of a `for loop` statement containing an `exit` statement.

The circuit generated by synthesizing the description from the previous example is shown in Figure 6.11. There is a copy of the contents of the loop for each possible iteration, that is, for every element of the vector. The `exit` statement is represented by the OR gates that select the multiplexers' inputs that do not contain the incrementer, for the current iteration and for all the remaining iterations, if the condition of the `exit` statement becomes true. It can be observed that the implementation of an `exit` statement in a loop is similar to that of an `if` statement within a loop.

6.1.7. Sequential `assert` Statement

The `assert` statement is useful for displaying warning or error messages during the simulation of a model. This statement tests the value of a Boolean condition and displays the message specified if the condition is false. The syntax of this statement is the following:

```
assert condition
   [report character_string]
   [severity severity_level];
```

The condition specified is an expression that must evaluate to a Boolean value. If this value is `TRUE`, the statement has no effect. If the value is `FALSE`, the text specified in the `report` clause is displayed.

The optional `report` clause may have as argument a character string of the predefined type `string`. If this clause is not specified, the character string displayed by default is `"Assertion violation"`.

The optional `severity` clause allows to specify the severity level of the assertion violation. The severity level must be an expression of the predefined type `severity_level`, defined in the `standard` package. This type contains the following values in the ascending order of the severity level: `note`, `warning`, `error`, and `failure`. If this clause is omitted, it is implicitly assumed to be `error`. The use of severity levels is described briefly as follows:

- The severity level `note` can be used to display information about the execution of the simulation.
- The severity level `warning` can be used in situations in which the simulation can be continued, but the results may be unpredictable.
- The severity level `error` is used when assertion violation represents an error that determines an incorrect behavior of the model; the simulation will be stopped.
- The severity level `failure` is used when assertion violation represents a fatal error, such as dividing by zero or addressing an array with an index that exceeds the allowed range. The simulation will be stopped.

Example 6.32

```
assert not (R = '1' and S = '1')
   report "Both signals R and S have value '1'"
   severity error;
```

When both signals `R` and `S` have the value '1', the specified message will be displayed and the simulation will be stopped.

To display a message unconditionally, the `FALSE` condition may be used, for example:

```
assert (FALSE) report "Start simulation";
```

In such cases, the VHDL '93 version of the language allows to use the `report` clause as a complete statement without the `assert` *`condition`* clause.

Notes

- The `assert` statement described in this section is a sequential statement, assuming that it appears in a process or subprogram. However, there is also a concurrent version of this statement, with the same format as the sequential version, but which can appear only outside a process or subprogram.
- Usually, synthesis tools ignore the `assert` statement.

6.2. Concurrent Statements

The operations in real systems are executed concurrently. The VHDL language models real systems as a set of subsystems that operate concurrently. Each of these subsystems may be specified as a separate process and communication between processes may be accomplished via signals. The complexity of each process may vary from a simple logic gate to a processor. The modeling of real systems in this form may be achieved with concurrent statements.

The most important concurrent statement is the process declaration. The processes were presented in Section 6.1.1, so only the main features of processes are presented here. Other concurrent statements are the concurrent signal assignment statement, the `block` statement, the concurrent `assert` statement, the concurrent procedure call statement, the component instantiation statement, and the `generate` statement. The procedure call statement is presented in Laboratory No. 9. The component instantiation and the `generate` statements are presented in Laboratory No. 8 dedicated to structural design.

6.2.1. Structure and Execution of an Architecture

As presented in Laboratory No. 6, an architecture definition has two parts: a declarative part and a statement part. In the declarative part objects that are internal to the architecture may be defined. The statement part contains concurrent statements which define the processes or interconnected blocks that describe the operation or the global structure of the system.

All processes in an architecture are executed concurrently with each other, but the statements within a process are executed sequentially. A suspended process is activated again when one of the signals in its sensitivity list changes its value. When there are multiple processes in an architecture, if a signal changes its value then all processes that contain this signal in their sensitivity lists are activated. The statements within the activated processes are executed sequentially, but independently from the statements in other processes.

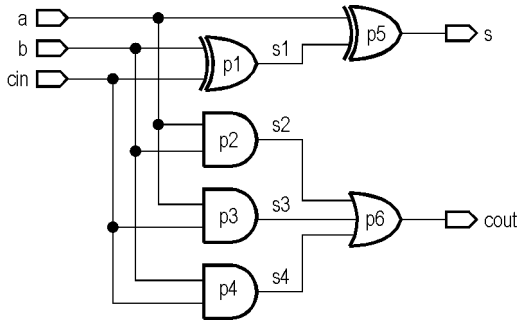

Figure 6.12. Logic diagram of a full adder.

Figure 6.12 presents the logic diagram of a full adder. In Example 6.33 each gate in this logic diagram is described by a separate process that is executed concurrently with other processes.

Example 6.33

```
entity add_1 is
   port (a, b, cin: in bit;
         s, cout: out bit);
end add_1;

architecture processes of add_1 is
   signal s1, s2, s3, s4: bit;
begin

   p1: process (b, cin)
   begin
      s1 <= b xor cin;
   end process p1;

   p2: process (a, b)
   begin
      s2 <= a and b;
   end process p2;

   p3: process (a, cin)
   begin
      s3 <= a and cin;
   end process p3;

   p4: process (b, cin)
   begin
      s4 <= b and cin;
   end process p4;

   p5: process (a, s1)
   begin
      s <= a xor s1;
   end process p5;

   p6: process (s2, s3, s4)
   begin
      cout <= s2 or s3 or s4;
   end process p6;

end processes;
```

Communication between processes can be achieved using signal assignment statements. These statements can be used both for activation of processes and synchronization between processes. For example, a signal can wait for an event on an input signal, which is assigned in another process. This signal is declared in the declarative part of the architecture and therefore it is visible for all processes within the architecture.

Note

- For communication between processes only signals can be used. Because variables are local objects in the process in which they are declared, variables cannot be used for communication between processes.

6.2.2. Processes

Processes are composed of sequential statements, but process declarations are concurrent statements. The process declaration has been presented in Section 6.1.1.1. The main features of a process are the following:

- It is executed in parallel with other processes;
- It cannot contain concurrent statements;
- It defines a region of the architecture where statements are executed sequentially;
- It must contain an explicit sensitivity list or a `wait` statement;
- It allows functional descriptions, similar to the programming languages;
- It allows access to signals defined in the architecture in which the process appears and to those defined in the entity to which the architecture is associated.

6.2.3. Concurrent Signal Assignment Statements

A concurrent signal assignment statement is equivalent to a process containing only that statement. Such a statement is executed in parallel with other concurrent statements or other processes. There are three types of concurrent signal assignment statements: simple signal assignment, conditional signal assignment, and selected signal assignment.

6.2.3.1. Simple Signal Assignment

This statement is the concurrent version of the sequential signal assignment statement and has the same form with this. As the sequential version, the concurrent assignment defines a new driver for the assigned signal. A concurrent assignment statement appears outside a process, within an architecture. A concurrent assignment statement represents a simplified form of writing a process and it is equivalent to a process that contains a single sequential assignment statement.

The description of the full adder from Example 6.33 may be simplified by using concurrent assignment statements, as shown in Example 6.34.

Example 6.34

```
entity add_1 is
   port (a, b, cin: in bit;
         s, cout: out bit);
end add_1;

architecture concurrent of add_1 is
   signal s1, s2, s3, s4: bit;
begin
      s1 <= b xor cin;
      s2 <= a and b;
      s3 <= a and cin;
      s4 <= b and cin;
      s  <= a xor s1;
      cout <= s2 or s3 or s4;
end concurrent;
```

As can be observed from the previous example, the concurrent assignment statements appear directly in the architecture, not inside a process. The order in which the statements are written is irrelevant. At simulation all statements are executed in the same simulation cycle.

In the case of processes, their activation is determined by the change of a signal in their sensitivity lists or by encountering a `wait` statement. In the case of concurrent assignment statements, the change of any signal

that appears in the right-hand side of the assignment symbol activates the assignment execution, without explicitly specifying a sensitivity list. Activation of an assignment statement is independent of activation of other concurrent statements within the architecture.

The concurrent assignment statements are used for dataflow descriptions. By synthesizing these statements, combinational circuits are obtained.

Note

- If there are several concurrent assignments to the same signal in an architecture, multiple drivers will be created for that signal. In these cases, either a predefined resolution function or a resolution function defined by the user must exist for the signal type. As opposed to concurrent assignments, if a process contains several sequential assignments to the same signal, only the last assignment will be effective.

6.2.3.2. Conditional Signal Assignment

The conditional assignment statement is functionally equivalent to the `if` conditional statement and has the following syntax:

```
signal <= [expression when condition else ...]
          expression;
```

The value of one of the source expressions is assigned to the target signal. The expression assigned will be the first one whose associated Boolean condition is true. When executing a conditional assignment statement, the conditions are tested in the order in which they are written. When the first condition that evaluates to value `TRUE` is encountered, its corresponding expression is assigned to the target signal. If none of the conditions evaluates to value `TRUE`, the expression corresponding to the `else` clause is assigned to the target signal.

The differences between the conditional assignment statement and the conditional `if` statement are the following:

- The conditional assignment statement is a concurrent statement, and therefore it can be used in an architecture, while the `if` statement is a sequential statement and can be used only inside a process.
- The conditional assignment statement can only be used to assign values to signals, while the `if` statement can be used to execute any sequential statement.

Example 6.35 defines an entity and two architectures for a two-input XOR gate. The first architecture uses a conditional assignment statement, while the second uses an equivalent `if` statement.

Example 6.35

```
entity xor2 is
   port (a, b: in bit;
         x: out bit);
end xor2;

architecture arch1_xor2 of xor2 is
begin
   x <= '0' when a = b else
        '1';
end arch1_xor2;

architecture arch2_xor2 of xor2 is
begin
   process (a, b)
   begin
      if a = b then x <= '0';
               else x <= '1';
      end if;
   end process;
end arch2_xor2;
```

The conditional signal assignment statement is implemented by a multiplexer which selects one of the source expressions. Figure 6.13 presents the circuit generated for the following statement:

```
s <= a xor b when c = '1' else
     not (a xor b);
```

The circuit in Figure 6.13 is generated initially by the synthesis tool, but the equality operator will be minimized later to a simple connection, so that signal `c` will control the multiplexer directly.

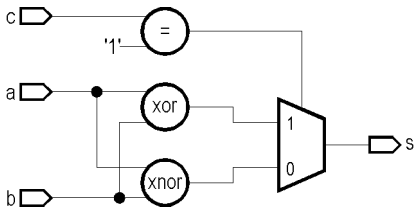

Figure 6.13. Synthesis of a conditional signal assignment statement.

The previous example is the simplest form of conditional signal assignment, with only one condition tested. Another example, in which two conditions are tested, is the following:

```
z <= a when s0 = '1' else
     b when s1 = '1' else
     c;
```

The conditions are evaluated in order, so that the first expression whose condition is true will be selected. This is equivalent in hardware terms to a series of two-way multiplexers, with the first condition controlling the multiplexer nearest to the output. The circuit for this example is illustrated in Figure 6.14. For this circuit the conditions have already been optimized, so that signals `s0` and `s1` control the multiplexers directly. From this circuit it can be seen that when `s0` is '1', then signal `a` is selected regardless of the value of `s1`. If `s0` is '0', then `s1` selects between the `b` and `c` inputs.

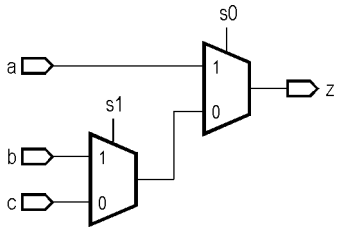

Figure 6.14. Synthesis of a two-way conditional signal assignment statement.

If there are a large number of branches, a long chain of multiplexers will result by synthesis. This aspect should be taken into account in the design: the later a source expression appears in the selection list, the more multiplexers the signals from this expression will pass through in the synthesized circuit.

Each condition in a conditional signal assignment statement is assumed to be independent of the others when synthesizing this statement. This means that, if the conditions are dependent (for example, they are based on the same signal), it is possible that no optimization will be performed. For example:

```
z <= a when sel = '1' else
     b when sel = '0' else
     c;
```

In this example, the second condition is dependent on the first. In fact, in the second branch, signal `sel` can only be '0'. Therefore, the second condition is redundant and the final `else` branch cannot be reached. This conditional signal assignment statement would still be implemented by two multiplexers, as illustrated in Figure 6.15.

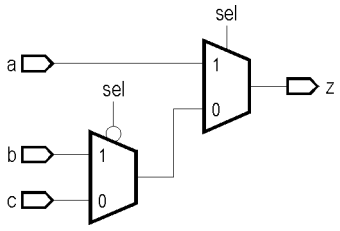

Figure 6.15. Synthesis of a conditional signal assignment statement with a redundant branch.

In the case of a simple example as the previous one, it is probable that the synthesis tool will eliminate the redundant multiplexer, but for more complex examples this cannot be guaranteed. The reason why an optimized implementation is not obtained is that, in the general case, detecting unreachable VHDL code is not a trivial task.

When the conditions are dependent on each other, it is more advantageous to use a selected signal assignment.

6.2.3.3. Selected Signal Assignment

Like the conditional signal assignment statement, the selected signal assignment statement allows to select a source expressions based on a condition. The difference is that the selected signal assignment statement uses a single condition to select between several options. This statement is functionally equivalent to the `case` sequential statement. The syntax is the following:

```
with selection_expression select
   signal <= expression_1 when options_1,
             ...
             expression_n when options_n,
             [expression when others];
```

The target signal is assigned the value of one of the expressions. The selected expression is the first from those expressions whose options include the value of the selection expression. The syntax of the options is the same as for the `case` statement. Thus, each option may be represented by an individual value or a set of values. If an option is represented by a set of values, either the individual values of the set may be specified, separated by the "|" symbol, or the range of values, or a combination of these. The type of the selection expression determines the type of each option.

All values from the selection expression's range must be covered by an option. The last option may be indicated by the `others` keyword, which specifies all the values from the selection expression's range that were not covered by the preceding options.

The following constraints exist for the various options:

- The values from the options cannot overlap each other.
- If the `others` option is missing, all the possible values of the selection expression must be covered by the set of options.

Note

- The options in the selected signal assignment statement are separated by commas.

In Example 6.36 the two-input XOR gate definition is modified to use a selected signal assignment statement. The equivalent form of this architecture using a `case` statement has been presented in Example 6.23.

Example 6.36

```
entity xor2 is
   port (a, b: in bit;
         x: out bit);
end xor2;

architecture arch_xor2 of xor2 is
   signal temp: bit_vector (1 downto 0);
begin
   temp <= a & b;
   with temp select
      x <= '0' when "00",
      x <= '1' when "01",
      x <= '1' when "10",
      x <= '0' when "11";
end arch_xor2;
```

6.2.3.4. `Block` Statement

A `block` statement defines a group of concurrent statements. This statement is useful to hierarchically organize the concurrent statements or to partition a list of structural interconnections in order to improve readability of the description. The syntax of the `block` statement is the following:

```
label: block [(guard_expression)]
   [declarations]
begin
   concurrent_statements
end block [label];
```

The mandatory label identifies the block. In the declaration part, local objects of the block may be declared. The possible declarations are those that may appear in the declarative part of an architecture:

- `Use` clauses;
- Port and generic declarations, as well as port map and generic map declarations;
- Subprogram declarations and bodies;
- Type and subtype declarations;
- Constant, variable, and signal declarations;
- Component declarations;
- File, attribute, and configuration declarations.

The order of concurrent statements in a block is not relevant, since all statements are always active. In a block other blocks may be defined, on several hierarchical levels. Objects declared in a block are visible in that block and in all internal blocks. When an object with the same name as another object in an outer block is declared in an inner block, the declaration in the inner block will be effective.

Example 6.37 illustrates the use of blocks on several hierarchical levels.

Example 6.37

```
B1: block
   signal s: bit;          -- declare "s" in block B1
begin
   s <= a and b;           -- "s" from block B1
   B2: block
      signal s: bit;       -- declare "s" in block B2
   begin
      s <= c and d;        -- "s" from block B2
      B3: block
      begin
         x <= s;           -- "s" from block B2
      end block B3;
   end block B2;
   y <= s;                 -- "s" from block B1
end block B1;
```

Introduction of blocks in a description does not affect the execution of the simulation model, but only has the purpose to organize the description.

In a block declaration a Boolean expression, called *guard expression*, may be specified. This expression, specified in parentheses after the `block` keyword, implicitly creates a Boolean signal named `guard`, which may be used to control the operations within the block. This signal may be read as any other signal inside the `block` statement, but no assignment statement may update it. Whenever a transaction occurs on any of the signals in a guard expression, the expression is evaluated and the `guard` signal is immediately updated. This signal takes on the value `TRUE` if the value of the guard expression is true and the value `FALSE` otherwise.

The `guard` signal may also be declared explicitly as a Boolean signal in the `block` statement. The advantage of this explicit declaration is that a more complex algorithm than that allowed by a Boolean expression may be used to control the `guard` signal. In particular, a separate process may be used to drive this signal.

If a guard expression is not specified in a block and the `guard` signal is not declared explicitly, then by default this signal is always `TRUE`.

The `guard` signal may be used to control the signal assignment statements inside a block. Such an assignment statement contains the `guarded` keyword after the assignment symbol, which determines a conditional execution of the assignment statement:

```
signal <= guarded expression;
```

This assignment is only executed if the `guard` signal of the block that contains the guard expression is true.

In Example 6.38, the signal `out1` will take on the value of `not in1` only when the value of the expression `clk'event and clk = '1'` will be true.

Example 6.38

```
rising_edge: block (clk'event and clk = '1')
begin
   out1 <= guarded not in1 after 5 ns;
   ...
end block rising_edge;
```

In Example 6.39, the `guard` signal is declared explicitly, so it can be assigned values like any other signal.

Example 6.39

```
UAL: block
   signal guard: boolean := FALSE;
begin
   out1 <= guarded not in1 after 5 ns;
   ...
   p1: process
   begin
      guard <= TRUE;
      ...
   end process p1;
end block UAL;
```

Notes

- In general, synthesis tools do not support guarded blocks. Such a block is equivalent to a process with a sensitivity list that contains conditional statements. Instead of a guarded block, a process with a sensitivity list may be used.
- Usually, simple blocks are ignored by synthesis tools.
- Although blocks may be used for design partitioning, the VHDL language allows a more powerful mechanism of partitioning, called component instantiation. This instantiation is presented in Laboratory No. 8.

6.2.3.5. Concurrent `assert` Statement

This statement is the concurrent version of the sequential `assert` statement, with the same syntax as its sequential version:

```
assert condition
   [report character_string]
   [severity severity_level];
```

The concurrent `assert` statement is executed whenever one of the signals in the conditional expression changes, as opposed to the sequential `assert` statement, which is executed when this statement is reached in a process or subprogram.

6.3. Examples of Combinational Circuits

6.3.1. Multiplexers

Multiplexers may be described using several methods. Example 6.40 describes the 4:1 multiplexer for 4-bit buses of Figure 6.16 using a selected signal assignment.

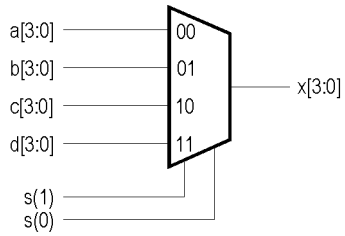

Figure 6.16. Block diagram of a 4:1 multiplexer for 4-bit buses.

Example 6.40

```
library ieee;
use ieee.std_logic_1164.all;
entity mux is
   port (a, b, c, d: in std_logic_vector (3 downto 0);
         s:          in std_logic_vector (1 downto 0);
         x:          out std_logic_vector (3 downto 0));
end mux;

architecture arch_mux of mux is
begin
   with s select
      x <= a when "00",
           b when "01",
           c when "10",
           d when "11",
           d when others;
end arch_mux;
```

The reason of using the `others` keyword is that the selection signal `s` is of type `std_logic_vector`, and there are nine possible values for a data object of this type. All the possible values of the selection signal must be covered. If the `others` option were not used, only four of the 81 values would be covered by the set of options. Other possible values of signal `s` are, for example, `"1X"`, `"UX"`, `"Z0"`, `"U-"`. For synthesis, `"11"` is the only meaningful value, but for simulation there are 77 other values that signal `s` may have. The metalogical value `"--"` may also be used to assign a don't care value to signal `x`.

The 4:1 multiplexer can be described with an `if` statement as shown in Example 6.41.

Example 6.41

```
architecture arch_mux of mux is
begin
mux4_1: process (a, b, c, d, s)
   begin
      if s = "00" then
         x <= a;
      elsif s = "01" then
         x <= b;
      elsif s = "10" then
         x <= c;
      else
         x <= d;
      end if;
   end process mux4_1;
end arch_mux;
```

Since the conditions imply mutually exclusive values of signal `s`, by synthesizing this description the same circuit is obtained as when a selected signal assignment statement is used. However, because the conditions contain a priority, the `if` statement is not advantageous when the conditions imply multiple signals that are mutually exclusive. Using an `if` statement in these cases may generate additional logic to ensure that the preceding conditions are not true. Instead of an `if` statement, it is more advantageous to use a Boolean equation or a `case` statement.

6.3.2. Priority Encoders

An example of a priority encoder is shown in Figure 6.17.

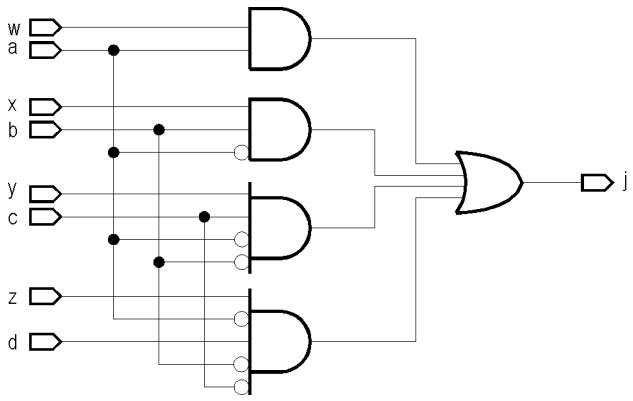

Figure 6.17. Logic diagram of a priority encoder.

This priority encoder may be described concisely with a conditional signal assignment statement, as in Example 6.42.

Example 6.42

```
library ieee;
use ieee.std_logic_1164.all;
entity priority_encoder is
   port (a, b, c, d: in std_logic;
         w, x, y, z: in std_logic;
         j: out std_logic);
end priority_encoder;

architecture priority of priority_encoder is
begin
   j <= w when a = '1' else
        x when b = '1' else
        y when c = '1' else
        z when d = '1' else
        '0';
end priority;
```

The `when-else` statement in the previous example indicates that signal `j` is assigned the value of signal `w` when `a` is '1', even if `b`, `c`, or `d` are '1'. Signal `b` holds priority over signals `c` and `d`, and signal `c` holds priority over signal `d`. If signals `a`, `b`, `c`, and `d` are mutually exclusive (that is, if it is known that only one will be asserted at a time), then the description of Example 6.43 is more appropriate.

Example 6.43

```
library ieee;
use ieee.std_logic_1164.all;
entity no_priority is
   port (a, b, c, d: in std_logic;
         w, x, y, z: in std_logic;
         j: out std_logic);
end no_priority;

architecture no_priority of no_priority is
begin
   j <= (a and w) or (b and x) or (c and y) or (d and z);
end no_priority;
```

The logic generated by synthesizing the description of Example 6.43 requires AND gates with only two inputs. Although using AND gates with more inputs in a CPLD (*Complex Programmable Logic Device*) does not

usually require additional resources, these gates could require additional logic cells and logic levels in an FPGA (*Field-Programmable Gate Array*) device. The descriptions of Example 6.42 and Example 6.43 are not functionally equivalent, however. This equivalence only exists if signals `a`, `b`, `c`, and `d` are known to be mutually exclusive. In this case, the description of Example 6.43 generates an equivalent logic with fewer resources.

6.4. Examples of Sequential Circuits

6.4.1. Synchronous and Asynchronous Sequential Circuits

Sequential circuits represent a category of logic circuits that include storage elements. These circuits contain feedback loops from the output to the input. The signals generated at the outputs of a sequential circuit depend on both the input signals and on the state of the circuit.

The present state of a sequential circuit depends on a previous state and on the values of input signals. In the case of synchronous sequential circuits, the change of state is controlled by a clock signal. With asynchronous circuits, the change of state may be caused by the random change in time of an input signal. The behavior of an asynchronous circuit is generally less secure, since the state evolution is also influenced by the delays of the circuit's components. The transition between two stable states may be attained by a succession of unstable, random states.

Synchronous sequential circuits are more reliable and have a predictable behavior. All storage elements of a synchronous circuit change their state simultaneously, which eliminates intermediate unstable states. By testing the input signals at well-defined times, the influence of delays and noises is reduced.

There are two techniques for designing sequential circuits: *Mealy* and *Moore*. In the case of Mealy sequential circuits, the output signals depend on both the current state and the present inputs. In the case of Moore sequential circuits, the outputs depend only on the current state, and they do not depend directly on the inputs. The Mealy method allows to implement a circuit by a minimal number of storage elements (flip-flops), but the possible uncontrolled variations of the input signals may be transmitted to the output signals. The design using the Moore method requires more storage elements for the same behavior, but the circuit operation is more reliable.

6.4.2. Flip-Flops

Example 6.44 describes a synchronous D-type flip-flop triggered on the rising edge of the clock signal (Figure 6.18).

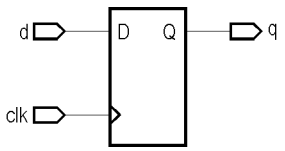

Figure 6.18. Symbol of a D-type flip-flop.

Example 6.44

```
library ieee;
use ieee.std_logic_1164.all;
entity dff is
   port (d:   in std_logic;
         clk: in std_logic;
         q:   out std_logic);
end dff;

architecture example of dff is
begin
   process (clk)
   begin
      if (clk'event and clk = '1') then
         q <= d;
      end if;
   end process;
end example;
```

The process used to describe the flip-flop is sensitive only to changes of the `clk` clock signal. A transition of the input signal `d` does not cause the execution of this process. The `clk'event` expression and the sensitivity list are redundant, because both detect changes of the clock signal. Some synthesis tools, however, will ignore the process sensitivity list, and thus the `clk'event` expression should be included to describe events triggered on the edge of the clock signal.

To describe a level-sensitive latch (Figure 6.19), the `clk'event` condition is eliminated and the data input `d` is inserted in the process sensitivity list, as shown in Example 6.45.

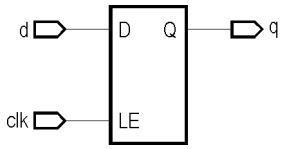

Figure 6.19. Symbol of a D-latch.

Example 6.45

```
architecture example of d_latch is
begin
   process (clk, d)
   begin
      if (clk = '1') then
         q <= d;
      end if;
   end process;
end example;
```

In Example 6.44 and 6.45 there is no `else` condition. Without this condition, an implied memory element is specified (that will keep the value of signal `q`). In other words, the following fragment:

```
if (clk'event and clk = '1') then
   q <= d;
end if;
```

has the same meaning for simulation as the fragment:

```
if (clk'event and clk = '1') then
   q <= d;
else
   q <= q;
end if;
```

This is consistent with the operation of a D-type flip-flop. Most synthesis tools do not allow to use an `else` expression after an `if (clk'event and clk = '1')` condition, because it may describe a logic for which the implementation is ambiguous.

As has been shown in Section 6.1.4.3, when a conditional statement does not have all the alternatives specified, a storage element (flip-flop) is generated by synthesis. In order to avoid to synthesize storage elements when they are not required, values must be assigned to variables or signals in each conditional branch.

6.4.3. Registers

Example 6.46 describes an 8-bit register by a process similar to that of Example 6.44, the difference being that `d` and `q` are vectors.

Example 6.46

```
library ieee;
use ieee.std_logic_1164.all;
entity reg8 is
   port (d:   in std_logic_vector (7 downto 0);
         clk: in std_logic;
         q:   out std_logic_vector (7 downto 0));
end reg8;
```

```
architecture ex_reg of reg8 is
begin
   process (clk)
   begin
      if (clk'event and clk = '1') then
         q <= d;
      end if;
   end process;
end ex_reg;
```

6.4.4. Counters

Example 6.47 describes a 3-bit counter.

Example 6.47

```
library ieee;
use ieee.std_logic_1164.all;
entity count3 is
   port (clk: in std_logic;
         count: buffer integer range 0 to 7);
end count3;

architecture count3_integer of count3 is
begin
   cnt: process (clk)
   begin
      if (clk'event and clk = '1') then
         count <= count + 1;
      end if;
   end process cnt;
end count3_integer;
```

In the previous example, the addition operator is used for the `count` signal, which is of type `integer`. Most of synthesis tools allow this use, converting the type `integer` to `bit_vector` or `std_logic_vector`. Nonetheless, using the type `integer` for ports poses some problems:

1) In order to use the value of `count` in another portion of a design for which the interface has ports of type `std_logic`, a type conversion must be performed.
2) The vectors applied during simulation of the source code cannot be used to simulate the model generated by synthesis. For the source code, the vectors should be integer values. The synthesized model will require vectors of type `std_logic`.

Because the native VHDL + operator is not predefined for the types `bit` or `std_logic`, this operator must be overloaded before it may be used to add operands of these types. The IEEE 1076.3 standard defines functions to overload the + operator for the following operand pairs: (`unsigned`, `unsigned`), (`unsigned`, `integer`), (`signed`, `signed`), and (`signed`, `integer`). These functions are defined in the `numeric_std` package of the 1076.3 standard.

Example 6.48 is the modified version of Example 6.47 in order to use the type `unsigned` for the counter's output.

Example 6.48

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity count3 is
   port (clk: in std_logic;
         num: buffer unsigned (3 downto 0));
end count3;

architecture count3_unsigned of count3 is
begin
   cnt: process (clk)
```

```
   begin
      if (clk'event and clk = '1') then
         count <= count + 1;
      end if;
   end process cnt;
end count3_unsigned;
```

Usually, synthesis tools supply additional packages to overload operators for the type `std_logic`. Although not standard packages, these are often used by designers because they allow arithmetic and relational operations on the type `std_logic`, and from this point of view they are even more useful than the `numeric_std` package. These packages do not require to use two additional types (`signed`, `unsigned`) in addition to `std_logic_vector`, as well as the functions to convert between these types. When using one of these packages for arithmetic operations, a synthesis tool will use an unsigned or signed (two's complement) representation for the type `std_logic_vector`, and will generate the appropriate arithmetic components as well.

6.4.5. Resetting Synchronous Logic

The previous examples do not make reference to resets or initial conditions in the logic described. The VHDL standard does not specify that a circuit must be reset or initialized. For simulation, the standard specifies that, unless a signal is explicitly initialized, it will be initialized to the value with the `'left` attribute of its type. To place the real circuits in a known state at initialization, reset and preset signals must be used.

Figure 6.20 illustrates a D-type flip-flop with an asynchronous reset signal. This flip-flop may be described as shown in Example 6.49.

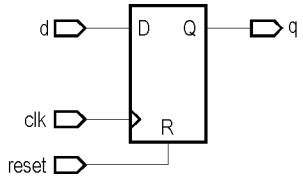

Figure 6.20. Symbol of a D-type flip-flop with asynchronous reset.

Example 6.49

```
architecture example_r of dff is          --  1
begin                                     --  2
   process (clk, reset)                   --  3
   begin                                  --  4
      if (reset = '1') then               --  5
         q <= '0';                        --  6
      elsif rising_edge (clk) then        --  7
         q <= d;                          --  8
      end if;                             --  9
   end process;                           -- 10
end example_r;                            -- 11
```

If the `reset` signal is asserted, then signal `q` will be assigned '0', regardless of the value of the clock signal. The `rising_edge` function is defined in the `std_logic_1164` package and detects the rising edge of a signal. This function can be used instead of the (`clk'event and clk = '1'`) expression if the `clk` signal is of type `std_logic`. In the same package is defined the `falling_edge` function as well, which detects the falling edge of a signal. These functions are preferred by some designers because at simulation the `rising_edge` function, for example, will ensure that the transition is from '0' to '1', and will not take into account other transitions such as from 'U' to '1'.

To describe a flip-flop with an asynchronous preset signal, lines 5-7 in the above example may be modified as follows:

```
      if (set = '1') then                 --  5
         q <= '1';                        --  6
      elsif rising_edge (clk) then        --  7
```

The flip-flops may also be reset (or preset) synchronously by including the appropriate condition inside the section of the process that is synchronous with the clock, as shown in Example 6.50.

Example 6.50

```
architecture example_r_sync of dff is
begin
   process (clk)
   begin
      if rising_edge (clk) then
         if (reset = '1') then
            q <= '0';
         else
            q <= d;
         end if;
      end if;
   end process;
end example_r_sync;
```

The process in the previous example is sensitive only to the clock signal. Synthesis produces a D-type flip-flop that is synchronously reset whenever the `reset` signal is asserted and a rising edge of the clock signal appears. Because most flip-flops in PLDs do not have synchronous sets or resets, implementing synchronous presets or resets requires using additional logic resources (Figure 6.21).

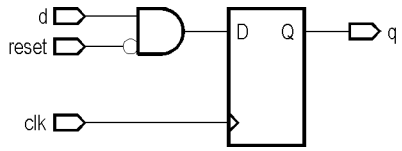

Figure 6.21. Additional logic resources required for a synchronous reset.

A combination of synchronous and asynchronous reset (or preset) may also be used. Sometimes two asynchronous signals are needed: a reset signal, as well as a set signal. Example 6.51 describes an 8-bit counter with asynchronous reset and set.

Example 6.51

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity count8 is
   port (clk:          in std_logic;
         reset, set:   in std_logic;
         enable, load: in std_logic;
         data:         in unsigned (7 downto 0);
         count:        buffer unsigned (7 downto 0));
end count8;

architecture arch_count8 of count8 is
begin
   cnt: process (reset, set, clk)
   begin
      if (reset = '1') then
         count <= (others => '0');
      elsif (set = '1') then
         count <= (others => '1');
      elsif (clk'event and clk = '1') then
         if (load = '1') then
            count <= data;
         elsif (enable = '1') then
            count <= count + 1;
         end if;
      end if;
   end process cnt;
end arch_num8;
```

In the previous example, both signals `reset` and `set` are used to asynchronously assign values to the counter registers. The combination of reset and preset signals in this example poses a problem related to synthesis. The `if-then-else` construct used in the process implies a precedence – that `count` should be assigned the value `"11111111"` only when signal `set` is asserted and `reset` is not asserted. The logic in Figure 6.22(a) assures this condition.

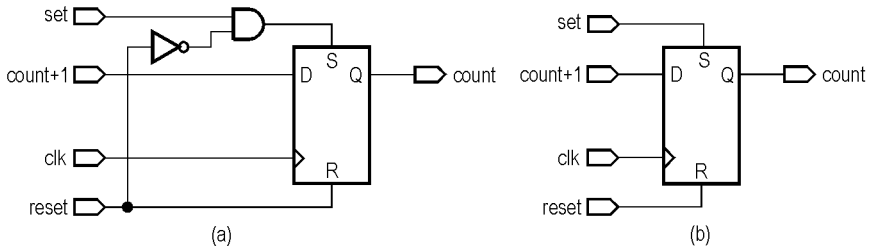

Figure 6.22. The synthesis result of Example 6.51: (*a*) additional logic assures that the `reset` signal is dominant; (*b*) the result assuming that the `reset` signal is dominant.

It is possible that this is not the intended behavior. Some synthesis tools recognize that this may not be the intended effect and flip-flops are by design either reset- or preset-dominant. Therefore, depending on the algorithm used by the synthesis software, the code of Example 6.51 generates the logic of either Figure 6.22(a) or Figure 6.22(b). Many CPLDs with product-term resets and presets are able to implement both variants. Likewise, most FPGA devices have the resources to implement product-term resets and presets. However, while most FPGA devices allow an efficient global reset or preset, most do not have the resources to provide an efficient product-term reset or preset, in which case the implementation of Figure 6.22(b) is preferred.

In all the preceding examples with reset or preset signals, either the `if` statement or the `rising_edge` function was used to describe synchronous circuits. These circuits may also be described with the `wait until` statement, but in this case the reset and preset signals must be synchronous. This is because for descriptions that are to be synthesized, the `wait` statement must be the first in the process, and therefore all the following statements will describe a synchronous logic.

6.4.6. Three-State Buffers and Bidirectional Signals

Most programmable-logic devices have three-state outputs or bidirectional I/O signals. Additionally, some devices have internal three-state buffers. The values that a three-state signal may have are `'0'`, `'1'`, and `'Z'`, all of which are supported by the type `std_logic`.

Example 6.52 presents the modified description for the counter of Example 6.51 to use three-state outputs. This counter does not have an asynchronous preset signal. This time the `std_arith` package is used and the type `std_logic_vector` for `data` and `count`.

Example 6.52

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_arith.all;
entity count8 is
   port (clk, reset:   in std_logic;
         enable, load: in std_logic;
         oe:           in std_logic;
         data:         in std_logic_vector (7 downto 0);
         count:        buffer std_logic_vector (7 downto 0));
end count8;

architecture arch_count8 of count8 is
   signal count_tmp: std_logic_vector (7 downto 0);
begin
   cnt: process (reset, clk)
   begin
      if (reset = '1') then
         count <= (others => '0');
      elsif rising_edge (clk) then
```

```
         if (load = '1') then
            count_tmp <= data;
         elsif (enable = '1') then
            count_tmp <= count_tmp + 1;
         end if;
      end if;
   end process cnt;

   oep: process (oe, count_tmp)
   begin
      if (oe = '0') then
         count <= (others => 'Z');
      else
         count <= count_tmp;
      end if;
   end process oep;

end arch_count8;
```

In this description two additional signals are used compared to the description of Example 6.51: `oe`, added to control the three-state outputs, and the local signal `count_tmp` declared in the architecture. The process labeled `oep` is used to describe the three-state outputs for the counter. If signal `oe` is not asserted, the outputs are placed in the high-impedance state. The `oep` process description is consistent with the behavior of a three-state buffer (Figure 6.23).

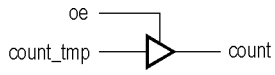

Figure 6.23. Three-state buffer.

The counter of the preceding examples may be modified to use bidirectional signals for its outputs. In this case, the counter may be loaded with the current value of its outputs, which means that the value loaded when the `load` signal is asserted will be the previous value of the counter or an external value, depending on the state of the `oe` signal.

In Example 6.53, the output enable of a three-state buffer is defined implicitly.

Example 6.53

```
mux: process (row_addr, col_addr, present_state)
begin
   if (present_state = row or present_state = RAS) then
      dram <= row_addr;
   elsif (present_state = col or present_state = CAS) then
      dram <= col_addr;
   else
      dram <= (others => 'Z');
   end if;
end process mux;
```

The three-state buffers of the `dram` signal are enabled if the value of `present_state` is `row`, `RAS`, `col`, or `CAS`. For any other values of the `present_state` signal, the output buffers are not enabled.

In the preceding examples, behavioral descriptions were used for three-state buffers. To generate these buffers, structural descriptions may be used as well, such as the `for generate` construct. This construct is described in Laboratory No. 8.

6.5. Applications

6.5.1. Identify and correct the errors in the following description:

```
library ieee;                                    --  1
use ieee.std_logic.all;                          --  2
entity t_c is                                    --  3
   port (clock, reset, enable: in bit;           --  4
```

```
         data: in std_logic_vector (7 downto 0);  --  5
         egal, term_cnt: out std_logic);          --  6
end t_c;                                          --  7
architecture t_c of t_c is                        --  8
   signal num: std_logic_vector (7 downto 0);     --  9
begin                                             -- 10
   comp: process                                  -- 11
   begin                                          -- 12
      if data = num then                          -- 13
         egal = '1';                              -- 14
      end if;                                     -- 15
   end process;                                   -- 16
                                                  -- 17
   count: process (clk)                           -- 18
   begin                                          -- 19
      if reset = '1' then                         -- 20
         num <= "111111111";                      -- 21
      elsif rising_edge (clock) then              -- 22
         num <= num + 1;                          -- 23
      end if;                                     -- 24
   end process;                                   -- 25
                                                  -- 26
   term_cnt <= 'z' when enable = '0' else         -- 27
               '1' when num = "1-------" else     -- 28
               '0';                               -- 29
end t_c;                                          -- 30
```

Use the *Active-HDL* system for a successful compilation of this description.

6.5.2. Analyze the examples presented for sequential statements. Simulate the descriptions of the equality comparator (Example 6.12), the counter (Example 6.21), the circuit to compare the bits of the same order of two vectors (Example 6.26), the circuit to count the number of ones of a vector (Example 6.27).

6.5.3. Write a sequence to set the signal `x` to the value of logical AND between all the lines of an 8-bit bus `a_bus[7:0]`.

6.5.4. Modify the following sequence to use a conditional signal assignment statement:

```
process (a, b, j, k)
begin
   if a = '1' and b = '0' then
      step <= "0100";
   elsif a = '1' then
      step <= j;
   elsif b = '1' then
      step <= k;
   else
      step <= "----";
   end if;
end process;
```

6.5.5. Transform the following sequence into a `case` statement:

```
with state select
   data <= "0000" when idle | terminate,
           "1111" when increase,
           "1010" when maintain,
           "0101" when decrease,
           "----" when others;
```

6.5.6. Transform the following sequence into two selected signal assignment statements:

```
case state is
   when idle => a <= "11"; b <= "00";
   when terminate | increase => a <= "01"; b <= "--";
   when maintain | decrease => a <= "10"; b <= "11";
   when others => a <= "11"; b <= "01";
end case;
```

6.5.7. Rewrite the following sequence using an `if` conditional statement:

```
output <= a when state = inactive else
          b when state = receive else
          c when state = transmit else
          d;
```

6.5.8. Simulate the combinational circuits described in Examples 6.40, 6.41, 6.42, and 6.43.

6.5.9. Build a 4-bit magnitude comparator with three outputs (equal, less than, and greater than) using:

a. Logical operators;
b. Relational operators;
c. A selected signal assignment statement;
d. A conditional `if` statement.

Compile the descriptions of the magnitude comparator and simulate their operation.

6.5.10. Describe an 8-bit 4:1 multiplexer using a `case` statement. The inputs to the multiplexer are `a[7:0]`, `b[7:0]`, `c[7:0]`, `d[7:0]`, `s[1:0]`, and the outputs are `q[7:0]`.

6.5.11. Design a BCD decoder for the 7-segment display. The inputs of the decoder are `bcd[3:0]`, and the outputs are `led[6:0]`.

6.5.12. Design a FIFO memory with a capacity of 8 words of 9 bits each. The block diagram of this FIFO memory is shown in Figure 6.24.

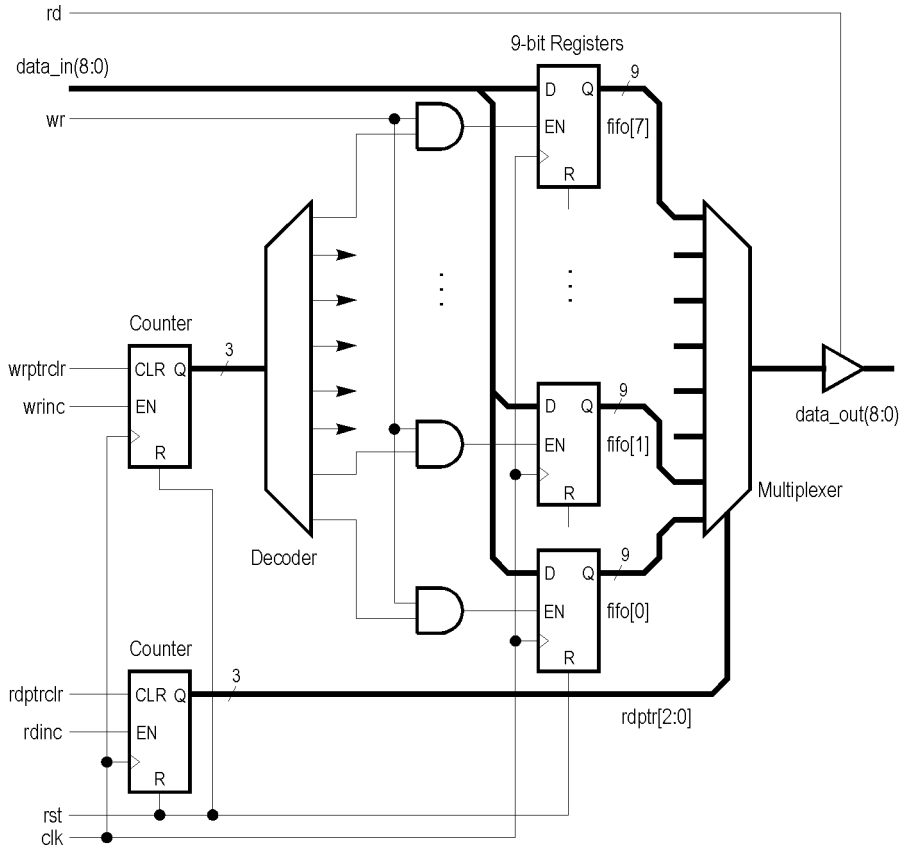

Figure 6.24. Block diagram of a FIFO memory.

When the read signal *rd* is asserted, the output *data_out* (9 bits) of the memory must be enabled. When the read signal is not asserted, the output must be placed in the high-impedance state. When the write signal *wr* is asserted, the value from the *data_in* input must be written into one of the 8 registers. The read and write pointers indicate which register to read and which register to write. To increment the read and write pointers, the *rdinc* and *wrinc* signals may be used, respectively. The *rdptrclr* and *wrptrclr* signals reset the read and write pointers to point the first register of the FIFO memory.

6.5.13. Compile the description of the 8-bit register of Example 6.46 and simulate the operation of this register.

6.5.14. Modify the description of the 8-bit register of Example 6.46 to use two additional input signals, `reset` and `init`. When the `reset` signal is asserted, the register will be reset to `"00000000"` asynchronously. When the `init` signal is asserted, the register will be set to `"11111111"` synchronously. Verify the operation of this register.

6.5.15. Compile the description of the 3-bit counter of Example 6.47 and verify the operation of this counter. Then modify the description to use the type `std_logic_vector` for the output vector `count`.

6.5.16. Modify the description of the 8-bit counter with three-state outputs of Example 6.52 to use a conditional assignment statement instead of the `oep` process. Add a three-state output signal named `collision`, which will be asserted when the signals `enable` and `load` are both active, provided that the outputs of the counter are activated by the `oe` signal.

6.5.17. Modify the description of the 8-bit counter of Example 6.52 to use bi-directional signals for its outputs. The counter will be loaded with its current outputs instead of the input vector `data`.

6.5.18. Design a binary multiplier for 8-bit numbers in sign-magnitude representation using the direct multiplication method. Use processes or concurrent signal assignment statements for each element of the multiplier.

6.5.19. Design a binary multiplier for 8-bit numbers in 2's complement representation using Booth's method. Use processes or concurrent signal assignment statements for each element of the multiplier.

6.5.20. Design a binary multiplier for 16-bit unsigned numbers using multiplication on groups of two bits. Use processes or concurrent signal assignment statements for each element of the multiplier.

6.5.21. Design a binary divider for numbers in sign-magnitude representation using the nonrestoring division method. The dividend is a 16-bit number, and the divisor is an 8-bit number. Use processes or concurrent signal assignment statements for each element of the divider.

6.5.22. Design a decimal multiplier for 4-digit numbers using the right-and-left-hand components method. Use processes or concurrent signal assignment statements for each element of the multiplier.

6.5.23. Describe in VHDL the architecture that uses horizontal microprogramming (Figure 4.5). Use processes or concurrent signal assignment statements for each element of the architecture.

6.5.24. Modify the description of the microprogrammed architecture in order to use vertical microprogramming (Figure 4.6).