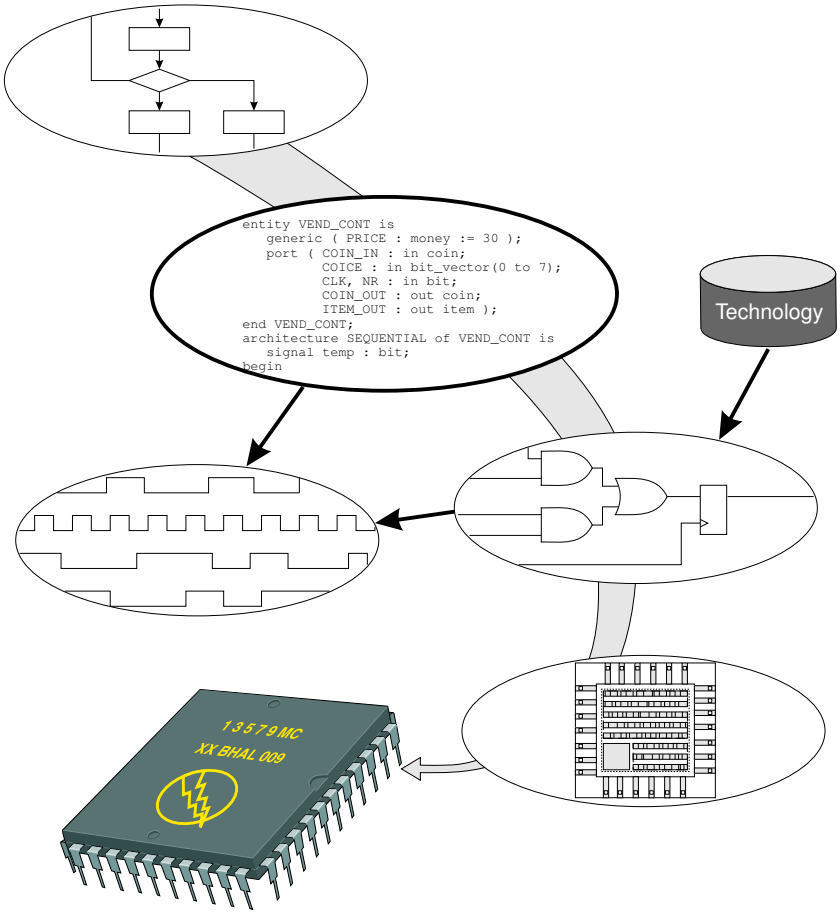
University of Ulm
Department of Microelectronics



VHDL Manual

Richard Geißler
Slavek Bulach
September 1998

Contents

1 Introduction

1.1 Motivation: IC Design Methodologies

In the last twenty years a change took place in the methodology of digital circuits design. In the past, integrated circuits were manually composed with graphical CAD-tools. For that purpose basic elements (logic gates from a library, or rather their symbols) had to be selected, placed on a schematic and connected with each other. In this way simple modules could be created which were then used to assemble complex circuits. This methodology is called bottom-up. It could take a long time to generate large circuits and the result was difficult to change because this meant laborious redrawing of the schematics.

Today, designing electrical systems deals with more and more complex systems, which can be integrated in single chips due to the increasing packing density. A short development cycle is another decisive factor designers have to consider in order to stay on top of the competition and to satisfy the requirements of the customers. Therefore, the reuse of once generated functional blocks and module in new systems is important. This requires a technology independent description of the circuits.

As far as digital circuits are concerned, the above considerations lead to the adoption of a top-down design flow. Using hardware description languages, modeling of systems at various levels of abstraction is possible. Due to the stepwise refinement in the top-down design flow, such a description language has to support all levels of abstraction: system specification, algorithmic description, functional blocks, and gate-level netlists. An important aspect in today´s design flows is the use of synthesis tools which automatically create gate-level netlists from behavioral descriptions. This requires a standardized language which would allow the simulation of the modeled system at different levels of abstraction.

VHDL (**V**HSIC **H**ardware **D**escription **L**anguage; VHSIC (**V**ery **H**igh **S**peed **I**ntegrated **C**ircuit)) meets all these requirements. It is possible to describe concurrent or sequential behavior of digital circuits, with or without timing, at various levels of abstraction. Hierarchical designs may be created by instantiating submodules and connecting them with each other. Nowadays, the language is supported by all major design tools mainly because it was standardized by the IEEE. Therefore, it can be used as an exchange medium between different CAD tools, or CAD tool users and chip vendors.

VHDL was initiated in the early 80's under the VHSIC program in the USA. The aim of this program was to develop a hardware description language for unambiguous documentation of digital systems. At that time, a number of companies designed VHSIC chips for the Department of Defense. Each used a different description language for developing and modeling their circuits. Data exchange, reuse and reproduction of designs was a big issue under these conditions. After the public release of VHDL in 1985 and additional enhancement in the following two years, VHDL was standardized by the IEEE in December 1987. It has also been recognized as an American National Standards Institute

(ANSI) standard. The official language description appears in the IEEE Standard VHDL Language Reference Manual (LRM). According to IEEE rules, it has to be reviewed every five years. This lead to the latest IEEE Standard, known as Std 1076-1993.

1.2 Contents and Structure of this Manual

This VHDL manual is based on the older IEEE Standard 1076-1987. The reason is that most of today´s design tools support this standard only and not the newer 1993 one.

The first part of the manual (Chapter 2) allows to cover the basics of VHDL in a very short time. This knowledge should be sufficient to describe simple digital circuits. The topics serve as a reference material for the introductory course (Design of Integrated Circuits with VHDL) lectures.

The second part, starting with Chapter 3, is based on the VHDL manual of the Group TECH, Department of Computer Science, University of Hamburg. It serves as a supporting documentation to the first part. The original version, written in German, and a lot of other useful information along with VHDL links can be found in the WWW under the following URLs:

http://tech-www.informatik.uni-hamburg.de/Dokumentation
http://tech-www.informatik.uni-hamburg.de/vhdl/vhdl.html

2 Basic VHDL Concepts

This chapter is intended to give a short introduction to the VHDL by presenting the basic concepts. Describing simple VHDL models should be possible after covering this material. Therefore, the chapter starts with an overview of the different components of VHDL models followed by a more detailed description in separate sections. Finally, the concepts of the VHDL library, analysis and simulation are presented.

The less exciting material on things like predefined types, allowed object identifiers and available operators for expressions, all of which are very similar to other programming languages like C or Pascal, is left out of this introductory chapter. This information, which has to be considered even in simple models, can be found in chapters 3, 4 and 5.

2.1 Components of a VHDL Model

The purpose of VHDL descriptions is to provide a model for digital circuits and systems. This abstract view of the real physical circuit is referred to as entity. An entity normally consists of five basic elements, or design units, shown in Figure 1.

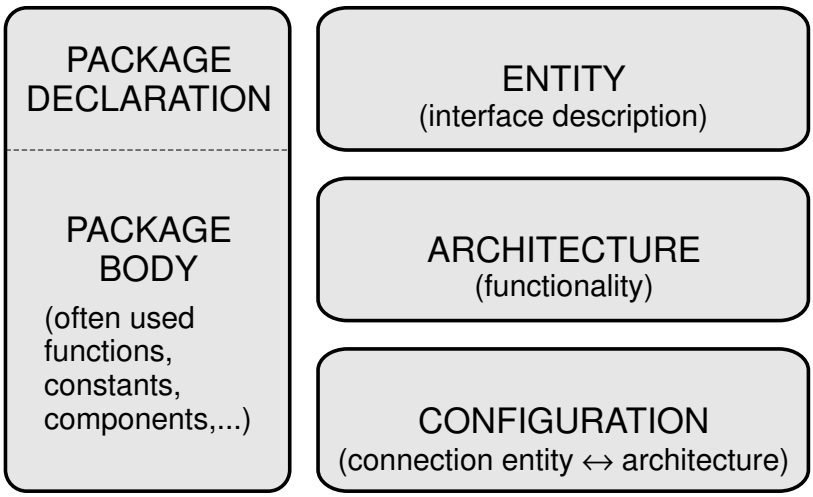


Figure 1: Basic elements of a VHDL model

In VHDL one generally distinguishes between the external view of a module and its internal description. The external view is reflected in the entity declaration which represents an interface description of a 'black box'. The important part of this interface description consists of signals over which the individual modules communicate with each other.

The internal view of a module and, therefore, its functionality is described in the architecture body. This can be achieved in various ways. One possibility is given by coding a behavioral description with a set of concurrent or sequential statements. Another possibility is a structural description which serves as a base for the hierarchically designed circuit architectures. Naturally, these two kinds

of architectures can also be combined. The lowest hierarchy level, however, must consist of behavioral descriptions. One of the major VHDL features is the capability to deal with multiple different architectural bodies belonging to the same entity declaration. In this case, it is necessary to bind one architecture to the entity in order to have a unique hierarchy for simulation or synthesis.

Being able to investigate different architectural alternatives permits the development of systems to be done in an efficient top-down manner. The ease of switching between different architectures has another advantage, namely, quick testing. This also includes switching between behavioral descriptions based on different algorithms, as well as switching to gate-level netlists, for example, after a partial synthesis is performed.

Which architecture should be used for simulation or synthesis in conjunction with a given entity is specified in the configuration section. If the architecture body consists of a structural description, then the binding of architectures and entities of the instantiated submodules, the so-called components, can also be fixed by the configuration statement.

The package is the last element mentioned here. It contains declarations of frequently used data types, components, functions, and so on. The package consists of a package declaration and a package body. The declaration is used, like the name implies, for declaring the above mentioned objects. This means, they become visible to other design units. In the package body, the definition of these objects can be carried out, for example, the definition of functions or the assignment of a value to a constant. Packages are language elements which can be compared with header files and the belonging codes, or object files, found in the programming language C. The partitioning of a package into its declaration and body provides advantages in compiling the model descriptions. This is further elaborated in section 2.7.

2.2 Entity Declaration

An entity declaration specifies the name of an entity and its interface. This corresponds to the information given by the symbols in traditional design methods based on drawing schematics. Signals which are used for communication with the surrounding modules are called ports.

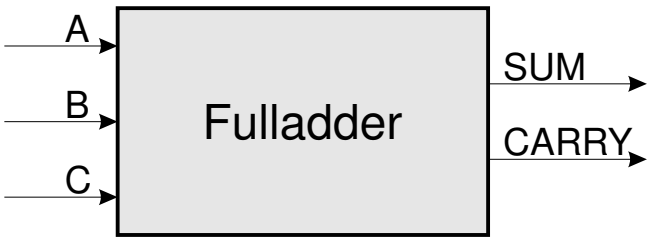


Figure 2: Interface of a full-adder module.

An entity declaration for the full-adder module shown in Figure 2 is as follows:

Example:
```
entity FULLADDER is
  -- (After a double minus sign (-) the rest of
  -- the line is treated as a comment)
  --
  -- Interface description of FULLADDER
  port (  A, B, C: in bit;
          SUM, CARRY: out bit);
end FULLADDER;
```

The module `FULLADDER` has five interface ports. Three of them are the input ports `A`, `B` and `C` indicated by the VHDL keyword `in`. The remaining two are the output ports `SUM` and `CARRY` indicated by `out`. The signals going through these ports are chosen to be of the type `bit`. This is one of the predefined types besides `integer`, `real` and others types provided by VHDL. The type `bit` consists of the two characters '0' and '1' and represents the binary logic values of the signals.

Every port declaration implicitly creates a signal with the name and type specified. It can be used in all architectures belonging to the entity in one of the following port modes:

Mode `in`: The port can only be read within the entity and its architectures.

Mode `out`: This port can only be written.

Mode `inout`: This port can be read and written. This is useful for modeling bus systems.

Mode `buffer`: The port can be read and written. Each port must have only one driver.

In order to improve the re-usability of VHDL codes these descriptions can be implemented with parameters, known as generics. For example, in a large hierarchical design it efficient to describe a register with an unconstrained bit width only once, and instantiate it in a structural description with the desired bit width specified by a generic. The entity declaration for such a register is given below:

Example:
```
entity DFF is
  -- parameter:  width of the data
  generic (width:  integer);
  -- input and output signals
  port (  CLK, NR: in bit;
          D: in bit_vector(1 to width);
          Q: out bit_vector(1 to width));
end DFF;
```

The parameter `width` affects the width of the input bus `D` and the output bus `Q`. These buses are declared as `bit_vector(1 to width)` which is equivalent to an array of signals (the number of elements in the array is specified by `width`) of the type `bit`, whose elements can be accessed by the index `1, 2, ..., width`.

In general, the entity declaration has the following format:

Syntax:
```
entity entity_name is
   [generics]
   [ports]
   [declarations (types, constants, signals)]
   [definitions (functions, procedures)]
[begin                                          -- normally not used
   statements]
end [entity_name];
```

A detailed description of generic and port declarations syntax is found in Section 2.3.3. The above examples can be used as templates. Besides the declaration of generics and ports, it is possible to declare types, constants, functions and signals which are accessible within the entity and the corresponding architectures. In the region between the keywords `begin` and `end` passive statements and procedures can be called. Passive means that no signal assignment neither in this procedure nor in any procedure called within the procedure is allowed. Passive statements could be used to generate warnings, for example, in case of setup and hold time violations on the inputs of a flip-flop.

2.3 Architecture

Following the entity declaration, the second important component of a VHDL description is the architecture. This is where the functionality and the internal implementation of a module is described. In general, a complex hierarchically structured system may have the topology shown in Figure 3.

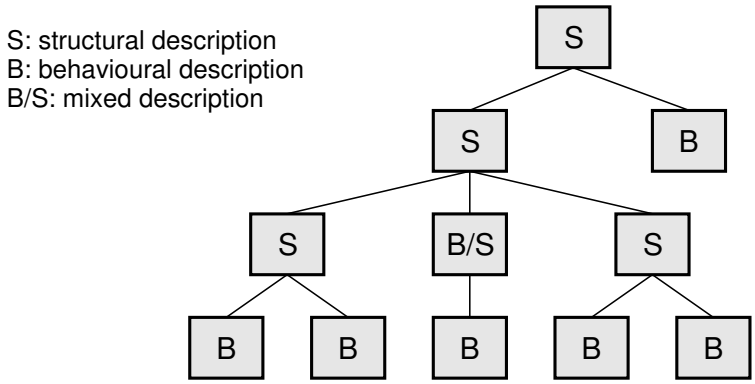


Figure 3: Hierarchical circuit design.

In order to describe such a system both behavioral and structural descriptions are required. A behavioral description may be of either concurrent or sequential type. Overall, VHDL architectures can be classified into the three main types:

- concurrent behavioral description
- sequential behavioral description
- structural description

All of these modeling styles share the same organization of an architecture.

Syntax:
```
architecture architecture_name of entity_name is
  [arch_declarative_part]

begin
  [arch_statement_part]
end [architecture_name];
```

As mentioned before, the architecture specifies the implementation of the entity entity_name. A label architecture_name must be assigned to the architecture. In case there are multiple architectures associated with one entity this label is then used within a configuration statement to bind one particular architecture to its entity. The architecture block consists of two parts: the arch_declarative_part before the keyword `begin` and the arch_statement_part after the keyword `begin`. In the declaration part local types, signals, components etc. are declared and subprograms are defined. The actual model description is done in the statement part. In contrast to programming languages like C, the major concern of VHDL is describing hardware which primary works in parallel and not in a sequential manner. For this reason, the statements in the arch_statement_part are executed concurrently, or in parallel. However, during the simulation of a VHDL description all concurrent statements are executed on a processor which processes all instructions sequentially. Therefore, a special simulation algorithm is used to achieve a virtual concurrent processing. This algorithm is explained in the following section.

2.3.1 Concurrent Behavioral Description

This kind of description specifies a dataflow through the entity based on concurrent signal assignment statements. A structure of the entity is not explicitly defined by this description but can be derived from it. As an example, consider the following implementation of the entity `FULLADDER` shown in Figure 2.

Example:
```
architecture CONCURRENT of FULLADDER is
begin
  SUM <= A xor B xor C after 5 ns;
  CARRY <= (A and B) or (B and C) or (A and C) after 3 ns;
end CONCURRENT;
```

Two concurrent signal assignment statements describe the model of the entity `FULLADDER`. The symbol `<=` indicates the signal assignment. This means that the value on the right side of the symbol is calculated and subsequently assigned to the signal on the left side. A concurrent signal assignment is executed whenever the value of a signal in the expression on the right side changes. In general, a change of the current value of a signal is called an *event*. Due to the fact that all signals used in this example are declared as ports in the entity declaration (see section 2.2) the arch_declarative_part remains empty.

Information about a possibly existing delay time of the modeled hardware is provided by the `after` clause. If there is an event on one of the inputs `A`, `B` or `C` at time T, the expression `A xor B xor C` is computed at this time T, but

the target signal (the output SUM) is scheduled to get this new value at time T + 5 ns. The signal assignment for CARRY is handled in exactly the same way except for the smaller delay time of 3 ns. If an explicit information about the delay time is missing then it is assumed to be 0 ns by default. This means that the signal assignment is executed immediately after an event on a signal on the right side is detected and the calculation of the new expression value is performed.

The simulation of concurrent signal assignments is explained with the help of a second example which gives an alternative implementation of the entity FULLADDER.

Example:
```
architecture CONCURRENT_VERSION2 of FULLADDER is
  signal PROD1, PROD2, PROD3 :  bit;
begin
  SUM <= A xor B xor C;                          -- statement 1
  CARRY <= PROD1 or PROD2 or PROD3;              -- statement 2
  PROD1 <= A and B;                              -- statement 3
  PROD2 <= B and C;                              -- statement 4
  PROD3 <= A and C;                              -- statement 5
end CONCURRENT_VERSION2;
```

A specification of delay time is missing in each of these signal assignments. Therefore, the delay time is set to 0 ns. Nevertheless, during the VHDL simulation the signal assignment is executed after an infinitesimally small delay time Δ, the so-called *delta-delay*. This is necessary to execute all concurrent signal assignment statements in virtually parallel fashion.

To observe what happens during VHDL simulation with concurrent signal assignments and to understand the *delta-delay* mechanism, assume an event on the input signal A which changes its value from a logical '0' to '1' at time T. For the values of other input signals assume a constant '0' on input B, and a constant '1' on input C. Due to the event on input A the statements 1, 3, and 5 are executed. The statement 1 results in a new value '0' for the signal SUM, the statement 3 leaves the signal PROD1 unchanged at '0', and the statement 5 calculates a change from '0' to '1' for PROD3. The new values get assigned at time T + Δ due to the missing explicit delay time information in the statements. The resulting event on PROD3 implicates that statement 2 has to be computed now. This causes a change of the signal CARRY from '0' to '1' which is assigned at time T + 2Δ. Due to this incremental computation with delta-cycles all concurrent statements are executed in virtually parallel manner. Figure 4 illustrates the sequence of signal changes during the simulation, starting with the event on A at time T and ending with the event on CARRY at T + 2Δ. Because no further events are scheduled for a third Δ, the system has stabilized and no more delta-cycles are necessary. All delta-cycles are hidden and do not appear on signal waveforms obtained during VHDL simulation. Signal waveforms show the state of signals before and after all delta-cycles associated with the simulation time T are executed.

The examples presented so far used only one kind of concurrent signal assignments. A set of additional concurrent statements is listed below. This list is not complete but it includes all statements necessary to describe simple VHDL

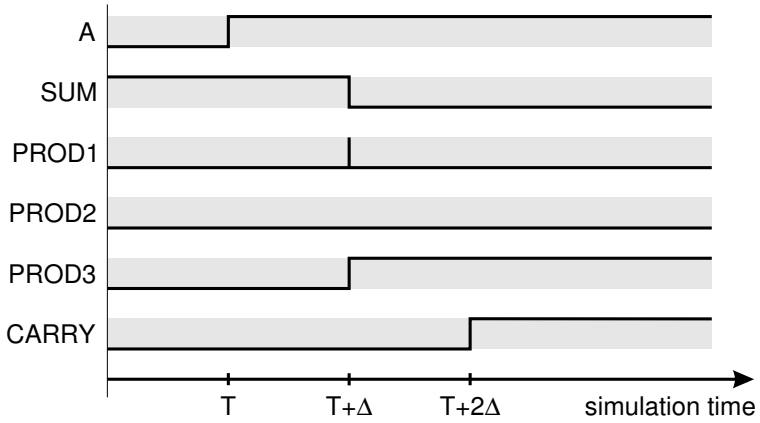


Figure 4: Simulation cycle with *delta-delay*; B = '0', C = '1'

models.

concurrent signal assignment statement: This statement is equal to the ones used in the previous examples.

Syntax:
```
[label:]
signal_name <= [transport] expression [after time_expr] {,
                           expression [after time_expr]};
```

Up to now the label was not used. With this element it is possible to assign a label to the statement which can be useful for documentation. Furthermore, it is possible to assign several events with different delay times to the target signal. In this case the values to be assigned and their delay times have to be sorted in ascending order. The keyword `transport` affects the handling of multiple signal events coming in short time one after another. This is explained in section 2.6.1.

conditional signal assignment statement: In this case there are different assignment statements related to one target signal. The selection of one assignment statement is controlled by a set of conditions condition. The conditional signal assignment statement can be compared with the well known `if - elsif - else` structure.

Syntax:
```
[label:]
signal_name <= expression when condition else
               {expression when condition else}
               expression;
```

Each time one signal either in expression or condition changes its value the complete statement is executed. Starting with the first condition, the first true one selects the expression which is computed and the resulting value is assigned to the target signal signal_name. To make the above syntax description more clear the optional statements `transport` and `after` time_expr are left out.

selected signal assignment statement: With this statement a choice between different assignment statements is made. The selection of the right assignment is done by the value of select_expression. The statement resembles a case structure.

Syntax:
```
[label:]
with select_expression select
  signal_name <= expression when value {,
                 expression when value};
```

assertion statement: This statement serves to generate warnings or error messages during simulation after testing a certain condition. It can be used, for example, to ensure the timing restrictions (setup, hold, ...) are met.

Syntax:
```
[assert_label:]
assert condition
  [report string_expr]
  [severity failure|error|warning|note];
```

If the test of the condition results in *false* then the message string_expr is displayed. Different `severity` levels of the generated message provide control over the VHDL simulator behavior. Most simulators allow to specify at which severity level the message is shown and at which level the simulation gets interrupted.

process statement: A process statement defines a region of code *within* all statements are executed *sequentially*. This concept is explained in detail in Section 2.3.2. Here it should be emphasized that every process statement as a whole is treated as a concurrent statement which is executed in parallel with all other concurrent statements.

2.3.2 Sequential Behavioral Description

Sequential behavioral descriptions are based on the process environment. As already mentioned, a process statement as a whole is treated as a concurrent statement within the architecture. Therefore, in the simulation time a process is continuously executed and it never gets finished. The statements *within* the process are executed sequentially without the advance of simulation time. To ensure that simulation time can move forward every process must provide a means to get suspended. Thus, a process is constantly switching between the two states: the execution phase in which the process is active and the statements within this process are executed, and the suspended state.

The change of state is controlled by two mutually exclusive implementations:

- With a list of signals in such a manner that an event on one of these signals invokes a process. This can be compared with the mechanism used in conjunction with concurrent signal assignment statements. There, the

statement is executed whenever a signal on the right side of the assignment operator <= changes its value. In case of a process, it becomes active by an event on at least one signal belonging to the *sensitivity list*. All statements between the keywords `begin` and `end process` are then executed sequentially.

Syntax:
```
[proc_label:]
process (sensitivity_list)
  [proc_declarativ_part]

begin
  [sequential_statement_part]
end process [proc_label];
```

The sensitivity_list is a list of signal names within round brackets, for example `(A, B, C)`.

- With `wait` statements in such a way that the process is executed until it reaches a `wait` statement. At this instance it gets explicitly suspended. The statements within the process are handled like an endless loop which is suspended for some time by a `wait` statement.

Syntax:
```
[proc_label:]
process
  [proc_declarativ_part]

begin
  [sequential_statements]
  wait ...;                    -- at least one wait statement
  [sequential_statements]
end process [proc_label];
```

The structure of a process statement is similar to the structure of an architecture. In the proc_declarativ_part various types, constants and variables can be declared; functions and procedures can be defined. The sequential_statement_part contains the description of the process functionality with ordered sequential statements.

An implementation of the full adder from Figure 2 with a sequential behavioral description is given below:

Example:
```
architecture SEQUENTIAL of FULLADDER is
begin
  process (A, B, C)
    variable TEMP : integer;
    variable SUM_CODE : bit_vector(0 to 3) := "0101";
    variable CARRY_CODE : bit_vector(0 to 3) := "0011";
  begin
    if A = '1' then TEMP := 1;
                else TEMP := 0;
    end if;
    if B = '1' then TEMP := TEMP + 1;
```

```
      end if;
      if C = '1' then TEMP := TEMP + 1;
      end if;      -- variable TEMP now holds the number of ones
      SUM <= SUM_CODE(TEMP);
      CARRY <= CARRY_CODE(TEMP);
    end process;
  end SEQUENTIAL;
```

The functionality of this behavioral description is based upon a temporary variable `TEMP` which counts the number of ones on the input signals. With this number one element, or one bit, is selected from each of the two predefined vectors `SUM_CODE` and `CARRY_CODE`. The initialization of these two vectors reflects the truthtable of a full-adder module.

The reason for this unusual coding is the attempt to explain the characteristics of a variable. A variable differs not only in the assignment operator (`:=`) from that of a signal (`<=`). It is also different with respect to time when the new computed value becomes valid and, therefore, readable to other parts of the model. Every variable gets the new calculated value *immediately*, whereas the new signal value is not valid until the beginning of the next delta-cycle, or until the specified delay time elapses.

If the above example had been coded with a signal as the temporary counter instead of the variable, then the correct functionality of this architecture as a full adder could not be ensured. After an event at time T on one of the input signals `A`, `B` or `C`, which are members of the `sensitivity_list`, the process is executed once. Now, assume that `TEMP` is declared as a *signal*. In the first `if` statement the signal `TEMP` is either reset to zero or in case `A = '1'` it is set to '1'. The assignment of the new value is scheduled for time $\mathbf{T} + \Delta$, which means that the appropriate event is written to an event queue for signal `TEMP`. The simulation continues with executing the second `if` statement at time **T** because computing a sequential statement does not advance the simulation time. Therefore, the signal `TEMP` still holds the same value it had before the process activation! This means that the intended counting of ones does not work with `TEMP` declared as *signal*.

In general, signal assignment statements within a process have to be handled with care, especially if the target signal will be read or rewritten in the following code before the process gets suspended (at the `wait` statement or, if a sensitivity list exists, at the end of the process). If this effect is taken into consideration, the process statement provides an environment in which a person familiar with programming languages like C or Pascal can easily generate a VHDL behavioral description. This remark, however, should not be understood that the process statement is there for people switching to VHDL. In reality, some functions can be implemented much more easily in a sequential manner. As an example, the implementation of a register belonging to the entity declaration on page 5 is shown:

Example:
```
architecture SEQUENTIAL of DFF is
begin
  process (CLK, NR)
  begin
```

```
      if (NR = '0') then
        -- Reset:  assigning "000...00" to the
        -- parameterized output signal Q
        Q <= (others => '0');
      elsif (CLK'event and CLK = '1') then
        Q <= D;
      end if;
    end process;
  end SEQUENTIAL;
```

Not explained until now is the use of attributes. In the above example, the attribute `CLK'event` is used to detect an edge on the `CLK` signal. This is equivalent to an event on `CLK`. The ability to detect edges on signals is based upon the storage of all events in event queues for every signal. Therefore, old values can be compared with the actual ones or even read. In contrast, variables always get the new assigned value immediately and the old value is not stored. Subsequently, during the simulation more memory is required for a signal for a variable. In complex system descriptions this fact should be taken into consideration.

Generally speaking, attributes exist not only in conjunction with signals. For instance, there are attributes associated with types and arrays. Some additional information on attributes is found in Section 7.4 on page 56.

Due to the similarity between sequential assignment statements in VHDL and common statements in other programming languages, only a brief description of their syntax is provided here.

sequential signal assignment statement: The syntax of a sequential signal assignment is very similar to the concurrent assignment statement, except for a label which can not be used.

Syntax: signal_name <= [transport] expression [after time_expr] {, expression after time_expr};

variable assignment statement: A variable assignment statement is very similar to a signal assignment. As already mentioned, a variable differs from a signal in that it gets its new value immediately upon assignment. Therefore, the specification of a delay time in a variable assignment is not possible. Attention must be paid to the assignment operator which is `:=` for a variable and `<=` for a signal.

Syntax: variable_name := expression;

assertion statement: Generating error or warning messages is possible also within the process environment. The syntax is nearly identical to a concurrent assertion statement, except for a label which can not be used.

Syntax:
```
assert condition
  [report string_expr]
  [severity failure|error|warning|note];
```

wait statement: This statements may only be used in processes without a sensitivity_list. The purpose of the `wait` statement is to control activation and suspension of the process.

Syntax:
```
wait [on signal_names]
     [until condition]
     [for time_expression];
```

The arguments of the `wait` statement have the following interpretations:

- `on` signal_names: The process gets suspended at this line until there is an event on at least one signal in the list signal_names. The signal_names are separated by commas; brackets are not used. It can be compared to the sensitivity_list of the process statement.
- `until` condition: The process gets suspended until the condition becomes true.
- `for` time_expression: The process becomes suspended for the time specified by time_expression.
- without any argument: The process gets suspended until the end of the simulation.

if-elsif-else statement: This branching statement is equivalent to the ones found in other programming languages and, therefore, needs no further explanation.

Syntax:
```
if condition then
  sequential_statements
{elsif condition then
  sequential_statements}
[else
  sequential_statements]
end if;
```

case statement: This statement is also identical to its corresponding equivalent found in other programming languages.

Syntax:
```
case expression is
  {when choices => sequential_statements}
  [when others => sequential_statements]
end case;
```

Either all possible values of expression must be covered with choices or the `case` statement has to be completed with an `others` branch.

null statement: This statement is used for an explicit definition of branches without any further commands. Therefore, it is used primarily in `case` statements, and also in `if` clauses.

Syntax:
```
null;
```

loop statement: is a conventional loop structure found in other programming languages.

Syntax:
```
[loop_label:]
while condition loop |                    --controlled by condition
for identifier in value1 to|downto value2 loop |  --with counter
loop                                              --endless loop
  sequential_statements
end loop [loop_label];
```

In the `for` loop the counter identifier is automatically declared. It is handled as a local variable within the loop statement. Assigning a value to identifier or reading it outside the loop is not possible.

exit and next statement: With these two statements a loop iteration can be terminated before reaching the keyword `end loop`. With `next` the remaining sequential statements of the loop are skipped and the next iteration is started at the beginning of the loop. The `exit` directive skips the remaining statements *and* all remaining loop iterations. In nested loops both statements skip the innermost enclosing loop if loop_label is left out. Otherwise, the loop labeled loop_label is terminated. The optional condition expression can be specified to determine whether or not to execute these statements.

Syntax:
```
next [loop_label][when condition];
exit [loop_label][when condition];
```

2.3.3 Structural Description

In structural descriptions the implementation of a system or model is described as a set of interconnected components, which is similar to drawing schematics. Such a description can often be generated with a VHDL netlister in a graphical development tool. Since there are many different ways to write structural descriptions, to explain all of them in one section would be more confusing than enlightening. Therefore, only one alternative approach is presented here.

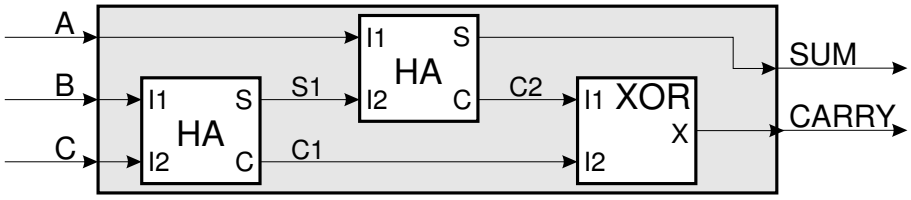


Figure 5: Structural implementation of a full adder.

As an introductive example, consider the implementation of a full-adder circuit shown in Figure 5. The corresponding entity declaration was discussed in Section 2.2 on page 5. The components `HA` and `XOR` are assumed to be predefined elements.

Example:
```
architecture STRUCTURAL of FULLADDER is
  signal S1, C1, C2 :  bit;
  component HA
    port (I1, I2 :  in bit; S, C : out bit);
  end component;
  component XOR
    port (I1, I2 :  in bit; X : out bit);
  end component;
begin
  INST_HA1 :  HA
    port map (I1 => B, I2 => C, S => S1, C => C1);
  INST_HA2 :  HA
    port map (I1 => A, I2 => S1, S => SUM, C => C2);
  INST_XOR : XOR
    port map (I1 => C2, I2 => C1, X => CARRY);
end STRUCTURAL;
```

In the declarative part of the architecture (the part between the keywords `is` and `begin`), all objects which are not yet known to the architecture have to be declared. In the example above, these are the signals (`S1`, `C1` and `C2`) used for connecting the components together, excluding the ports of the entity `FULLADDER`. In addition, the components `HA` and `XOR` have to be declared. The declaration of a component consists of declaring its interface ports and generics to the actual model.

Often used components could be selected from a library of gates defined in a package and linked to the design. In this case the declaration of components usually is done in the package, which is visible to the entity. Therefore, no further declaration of the components is required in the architecture declarative part.

The actual structural description is done in the statement part of the architecture (between the keywords `begin` and `end` arch_name) by the instantiation of components. The components' reference names `INST_HA1`, `INST_HA2` and `INST_XOR`, also known as instance names, must be unique in the architecture. The port maps specify the connections between different components, and between the components and the ports of the entity. Thus, the components' ports (so-called locals) are mapped to the signals of the architecture (so-called actuals) including the signals of the entity ports. For example, the input port `I1` of the half adder `INST_HA1` is connected to the entity input signal `B`, input port `I2` to `C`, and so on.

The instantiation of a component is a concurrent statement. This means that the order of the instances within the VHDL code is of no importance.

Syntax: component declaration:
```
component component_name
  [generic ( generic_list:  type_name [:= expression] {;
             generic_list:  type_name [:= expression]} );]
  [port (    signal_list:  in|out|inout|buffer type_name {;
             signal_list:  in|out|inout|buffer type_name} );]
end component;
```

component instantiation:

```
component_label:   component_name
    port map (signal_mapping);
```

The syntax of a component declaration statement consists of a general specification of generics and ports which were discussed in Section 2.2 in reference to the entity declaration. The connection of the architecture's signals to the ports of the components can be done in various ways. The syntax used in the above example makes the assignment in the following way:

Syntax: signal_mapping: declaration_name => signal_name

It is important to note that the symbol '=>' is used within a port map in contrast to the symbol '<=' used for concurrent or sequential signal assignment statements!

2.4 Configuration Declaration

The concept of configuration in VHDL allows an entity to have multiple associated architectures. The role of the configuration declaration is to define a unique system description from the various design units.

2.4.1 Configuration of Behavioral Descriptions

In this case, the generation of a configuration declaration is very simple. The only information which the configuration has to include is the choice of one architecture for the given entity. This binding is established by naming the entity architecture pair as follows:

Syntax:

```
configuration configuration_name of entity_name is
  for architecture_name
  end for;
end configuration_name;
```

The configuration_name can be any name. It is allowed to have more than one configuration for one entity such that for every architecture an appropriate configuration exists.

For the behavioral descriptions of the full adder two configurations are specified:

Example:

```
configuration CFG_ONE of FULLADDER is
  for CONCURRENT
  end for;
end CFG_ONE;

configuration CFG_TWO of FULLADDER is
  for SEQUENTIAL
  end for;
end CFG_TWO;
```

The first configuration CFG_ONE binds the concurrent behavioral description CONCURRENT to the entity FULLADDER. The second configuration CFG_TWO selects the sequential behavioral description SEQUENTIAL.

2.4.2 Configuration of Structural Descriptions

If the configuration binds a structural description to an entity then further information about the instantiated components is required. Due to the fact that the name of a component in the component declaration needs not be the same as the entity name of the instantiated component, their binding must be done by the configuration. Furthermore, the binding of the component's entity and architecture must be established by the configuration.

The following example illustrates the use of configuration statements in a structural description. It refers to the architecture `STRUCTURAL` of the full-adder circuit in Section 2.3.3, page 15. Assuming that the entities `HALFADDER` and `XOR2D1` exist for the instantiated components `HA` and `XOR` respectively, and that the architecture `CONCURRENT` is provided for each of these entities, then the configuration may look like the following:

Example:
```
configuration THREE of FULLADDER is
  for STRUCTURAL
    for INST_HA1, INST_HA2:  HA
      use entity WORK.HALFADDER(CONCURRENT);
    end for;
    for INST_XOR: XOR
      use entity WORK.XOR2D1(CONCURRENT);
    end for;
  end for;
end THREE;
```

The first and second line contain the configuration name and the binding of the entity and architecture. Since the architecture `STRUCTURAL` is a structural description, the components `HA` and `XOR` have to be configured. This is done by the two inner `for`-loops. The first loop specifies that for the two instances `INST_HA1` and `INST_HA2` of the component `HA` an entity `HALFADDER` with its architecture `CONCURRENT` has to be used. In addition, it is stated that the `HALFADDER` is taken from the library `WORK` (more about libraries in Section 2.7).

In general, a configuration declaration belonging to an architecture with instantiated components is of the form:

Syntax:
```
configuration configuration_name of entity_name is
  for architecture_name
    for label|others|all:  comp_name
      use entity [lib_name.]comp_entity_name(comp_arch_name) |
      use configuration [lib_name.]comp_configuration_name
        [generic map (...)]
        [port map (...)] ;
    end for;
    ...
  end for;
end configuration_name;
```

If there exists a configuration for the entity of the instantiated component, then the binding of the entity and architecture is already done by this configuration. Therefore, it is possible to refer to the configuration of the submodule by using the statement `use configuration ...` instead of defining the

entity-architecture pair explicitly. Furthermore, the mapping of generic and port names between the component declaration (so-called locals)and the entity declaration of the submodule (so-called formals)can be done by the `generic map` and `port map` statements. This can be useful if the order of the ports or generics is different or rearranging the order on purpose or locals and formals are different.

2.5 Packages

A packages is used as a collection of often used datatypes, components, functions, and so on. Once these object are declared and defined in a package, they can be used by different VHDL design units. In particular, the definition of global information and important shared parameters in complex designs or within a project team is recommended to be done in packages.

It is possible to split a package into a declaration part and the so-called body. The advantage of this splitting is that after changing definitions in the package body only this part has to be recompiled and the rest of the design can be left untouched. Therefore, a lot of time consumed by compiling can be saved.

2.5.1 Package Declaration

As the name implies, a package declaration includes all globally used declarations of types, components, procedures and functions. A possible package declaration is presented by means of an example:

Example:
```
package MY_PACK is
  type SPEED is (STOP, SLOW, MEDIUM, FAST);
  component HA
    port (I1, I2 :  in bit; S, C : out bit);
  end component;
  constant DELAY_TIME : time;
  function INT2BIT_VEC (INT_VALUE : integer)
    return bit_vector;
end MY_PACK;
```

The name of this package is `MY_PACK`. It consists of different declarations, such as a type `SPEED`, a component `HA`, and so on. Attention should be paid to the declaration of the constant `DELAY_TIME` and the function `INT2BIT_VEC` which are declared but are not defined. Their definitions will be done in the package body but it would be possible to define the constant `DELAY_TIME` in the package declaration part as well. The definition of functions must be done in a package body.

If the above package had been compiled into the library `MY_LIB` (Section 2.7) then the following statements were also needed in the VHDL model which uses this package:

Example:
```
library MY_LIB;
use MY_LIB.MY_PACK.all;
entity EXAMPLE is
```

...

The `library` statement makes the library `MY_LIB` accessible for the following VHDL description. With the subsequent `use` statement all elements (indicated by the keyword `all`) from the package `MY_PACK` are included in the entity of the module `EXAMPLE`.

2.5.2 Package Body

In the package body the definition of functions and procedures that were *only* declared in the package declaration must be specified. Constants which were declared only must get a value assigned to them in the package body.

The body of the package `MY_PACK` could be defined as:

Example:
```
package body MY_PACK is
  constant DELAY_TIME : time := 1.25 ns;
  function INT2BIT_VEC (INT_VALUE : integer)
    return bit_vector is
  begin
    -- sequential behavioral description (omitted here)
  end INT2BIT_VEC;
end MY_PACK;
```

The binding between the package declaration and the body is established by using the same name. In the above example it is the package name `MY_PACK`.

2.5.3 Important Packages

There are four important packages often used in VHDL descriptions.

STANDARD: The package STANDARD is usually integrated directly in the simulation or synthesis program and, therefore, it does not exist as a VHDL description. It contains all basic types: boolean, bit, bit_vector, character, integer, and the like. Additional logical, comparison and arithmetic operators are defined for these types within the package.

The package STANDARD is a part of the `STD` library. Thus, it does not have to be explicitly included by the `use` statement.

TEXTIO: The package TEXTIO contains procedures and functions which are needed to read from and write to text files.

This package is also a part of the library `STD`. It is *not* included in every VHDL description by default. Therefore, if required, it has to be included by the statement `use STD.TEXTIO.all;`.

STD_LOGIC_1164: The STD_LOGIC_1164 package has been developed and standardized by the IEEE. It introduces a special type called `std_ulogic` which has nine different logic values. The reason for this enhancement is that the type `bit` is not suitable for the precise modeling of digital circuits due to the missing values, such as `uninitialized` or `high impedance`.

The type `std_ulogic` consists of the following elements:

Declaration:
```
type std_ulogic is (
                    'U',    -- uninitialized
                    'X',    -- forcing unknown
                    '0',    -- forcing 0
                    '1',    -- forcing 1
                    'Z',    -- high impedance
                    'W',    -- weak unknown
                    'L',    -- weak 0
                    'H',    -- weak 1
                    '-' );  -- "don't care"
```

Besides this type used for modeling single wires other types are declared in the STD_LOGIC_1164 package. Frequently used in descriptions of bus systems are the types `std_ulogic_vector` and `std_logic_vector`. In addition, the also includes the definitions of resolution functions (see Section 2.6.2) and simple boolean functions.

The use of the types `std_ulogic` and `std_logic` is strongly recommended. The package STD_LOGIC_1164, if it is available on the system installation, is usually be kept in the logical library `IEEE`. It could be referenced with the two statements:

Syntax:
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
```

STD_LOGIC_ARITH or NUMERIC_STD: Two additional packages, STD_LOGIC_ARITH (provided by SYNOPSYS) and NUMERIC_STD (provided by the IEEE), represent an additional part for the STD_LOGIC_1164 package. They contain basic arithmetic functions to enable calculations and comparisons based on the types `std_ulogic_vector` and `std_logic_vector`. These types represent buses – a bunch of signal lines – whose state can be interpreted as a binary or as a two's complement number. Therefore, it is necessary to specify which number representation is valid for a given bus system. This can be done by a conversion into the data types `unsigned` and `signed`. The appropriate conversion functions are also defined in these packages.

Example:
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
architecture DETAILED of EXAMPLE is
  signal A, B :   std_logic_vector (7 downto 0);
  signal SUM :    std_logic_vector (8 downto 0);
  signal SUM_S :  signed (8 downto 0);
  signal PROD :   std_logic_vector (15 downto 0);
  signal PROD_S : signed (15 downto 0);
begin
  -- extension by one digit, conversion into a two's
  -- complement number and calculation of the sum:
  SUM_S <= signed(A(7) & A) + signed(B(7) & B);
  -- conversion to 9 bit std_logic_vector:
```

```
    SUM <= conv_std_logic_vector(SUM_S, 9);
    -- calculation of the product:
    PROD_S <= signed(A) * signed(B);
    -- conversion to 16 bit std_logic_vector:
    PROD <= conv_std_logic_vector(PROD_S, 16);
  end DETAILED;
```

In the above example the sum and the product of the two busses `A` and `B` are calculated. Because the width of the resulted sum is the same as those of the operands, the width of `A` and `B` has to be extended by one bit in order to avoid an overflow. Since both `A` and `B` are two's complement numbers their MSB's have to be doubled. This is achieved by the catenations `A(7) & A` and `B(7) & B`. After converting signals `A` and `B` with the `signed(...)` and adding, the result is assigned to a temporary signal `SUM_S`. This signal is then converted back to a **9** bit wide bus of the type `std_logic_vector` with the function `conv_std_logic_vector(SUM_S, 9)`. For the multiplication, the width of the result is 16 bit, which is equal to the sum of the widths of the operands `A` and `B`. The appropriate information is required in the conversion of `PROD_S` to `PROD`.

2.6 Additional Signal Characteristics

As already mentioned in Section 2.3.2, signals differ from the concept of VHDL variables, as well as variables found in other programming languages. In the following section additional important characteristics of signals are presented.

2.6.1 Delay Models

During the VHDL simulation event queues, which contain all future signal events, are created and manipulated. The handling of new generated events and the events already existing in the event queues is influenced by different delay models. Based on the so-called preemption mechanism, actions already existing in the event queue are partially removed when a new event is scheduled.

The following two delay models are distinguished:

transport delay model: All entries, which are scheduled for the same or later time in the event queue, are deleted. This delay model is specified by the keyword `transport` in the signal assignment statement.

inertial delay model: The inertial delay model is the default and the most commonly used one. In this model all entries of the event queue which would be deleted by the transport delay model are also removed. In addition, the following rules are applied:

1. Mark the entry directly before the new one if it has the same value.
2. Mark the current and the new entry.
3. Delete all entries which are *not* marked.

The consequence of this delay model is that all signal impulses shorter than the delay specified in the signal assignment statement are swallowed.

Figure 6 illustrates the difference between the two delay models with an example. The starting point is a signal `SIG` which consists of impulses with different durations. Signals `SIG_T` and `SIG_I` are generated from the signal `SIG` by the following two statements:

Statement:
```
SIG_T <= transport SIG after 3 ns;
SIG_I <=           SIG after 3 ns;
```

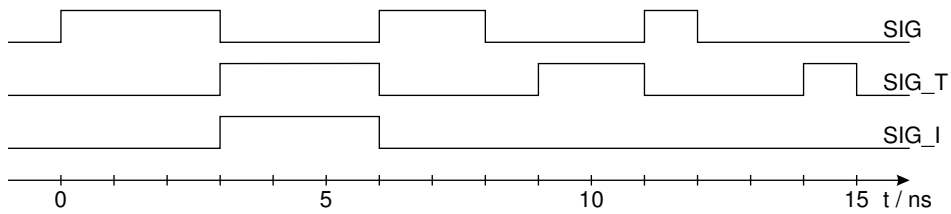


Figure 6: Waveforms generated by different delay models.

Due to the specified delay time of 3 ns all impulses shorter than 3 ns are filtered out in case of the inertial delay model. This is illustrated by the signal waveform `SIG_I`. This behavior is similar to real hardware in which charging an output node of a gate needs some time and, therefore, a spike or glitch cannot propagate through logic gates.

The assumption that a spike does not propagate through a gate if its duration is less than the delay time of the gate is sometimes not accurate enough. Therefore, a third delay model was defined in the newer 1993 VHDL standard in which the delay time and the maximum filtered pulse width can be specified separately.

2.6.2 Resolution Functions

Another interesting feature of signals is that *multiple* signal assignment statements may write onto *one* signal. This means that there exists more than one driver for such a signal. This feature is necessary for modeling bus systems where normally more than one module can write to a bus. In these cases, the resolution function is used to calculate the resulting value of the signal depending on the values written from all drivers.

The STD_LOGIC_1164 package contains the definition of such a resolution function for the type `std_ulogic` (**u** = unresolved) named `resolved`. Within the declaration of the subtype `std_logic` an implicit call to the resolution function `resolved` is defined. Consequently, every new value assigned to a signal of type `std_logic` first goes through the resolution function. The function calculates the real value of the target signal taking into consideration the values produced by the remaining drivers.

2.7 Analysis of VHDL Models

Once the VHDL description of an electronic system is complete, the next step within the design flow is to simulate the system in order to verify the correct functionality. The second step is to synthesize a gate level netlist for the target technology. In both cases the VHDL models have to be analyzed, which is similar to the compilation in other programming languages like C. From the circuit developer's point of view, the most important action during the analysis process is the checking of the used syntax. The output data generated by this process are stored in design libraries.

A design library is a directory on the computer system where certain analyzed designs are stored. Due to the fact that the path to this physical location depends on the installation of the VHDL system and is usually different for different hosts, *logical* library names (STD, IEEE, etc.) are used in VHDL descriptions. The mapping of logical library names and the physical storage location is done by the system administrator after installation of the VHDL system. In addition, each user can create personal libraries but then he is responsible for the proper mapping. The concept of logical libraries ensures that VHDL codes are portable. It is highly useful for the exchange of design data.

Many different design libraries may exist simultaneously. Only one of them can be used as the actual working library. The logical name of this library is WORK. Which library is actually used as the design library is defined by settings of the system. During the analysis process all design units are stored in the library WORK. Figure 7 illustrates the library concept.

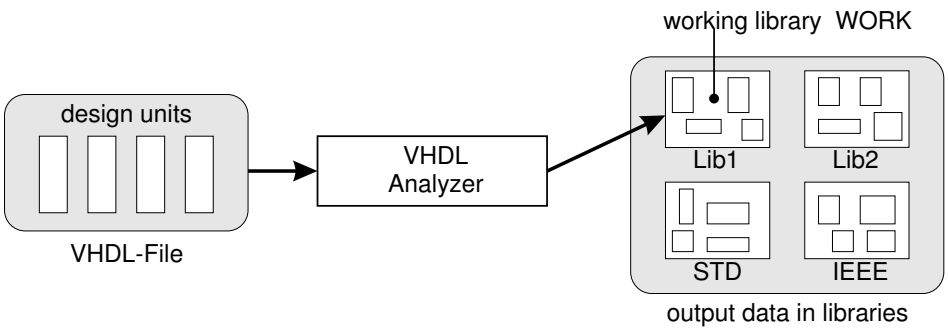


Figure 7: Analysis process.

During the analysis of VHDL models it is important to follow the proper order. Hierarchical systems have to be analyzed starting from the bottom level to the top. In addition, different design units have to be analyzed in the order shown in Figure 8.

2.8 Simulation

After the successful analysis of VHDL models, their simulation could be performed to verify the correct functionality. For this purpose, the elements in the lowest hierarchy level must be available as behavioral descriptions. Starting

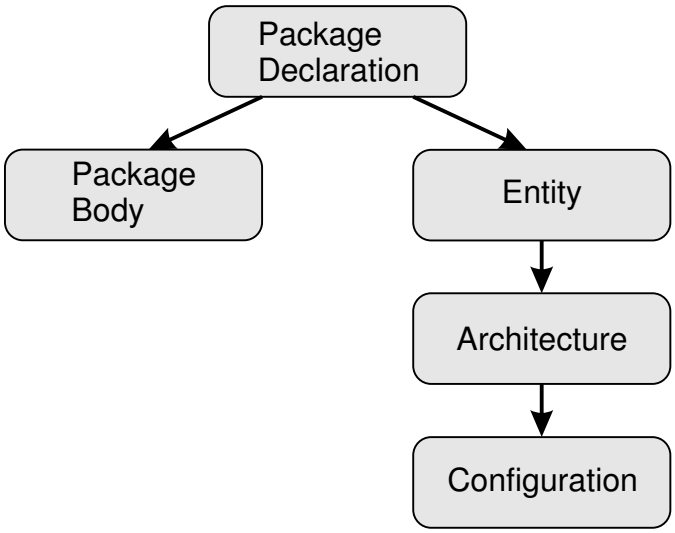


Figure 8: Dependencies during analysis.

point of the simulation is the analyzed configuration declaration of a testbench or the top-level module.

Before the actual simulation takes place, the following two step are executed (without the interaction between the circuit developer and the simulation tool):

1. Elaboration phase: The most important part of this step is assembling the hierarchy. This is where all entity-architecture pairs are built as specified by the configurations. This is similar to the activities taking place during linking in other programming languages like C. Furthermore, memory is allocated for signals, variables and constants, and their values are initialized as specified.

2. Initialization phase: All processes are executed once until they get suspended by the first encountered `wait` statements, or after one complete pass in case of an existing sensitivity_list. Signals are assigned their starting values and the simulation time is set to zero.

The simulation is usually done by stimulating the input signals of the unit under test (UUT) with the appropriate waveforms. This is easily achieved by the so-called testbench, a special entity which resides on top of the complete unit under test. The testbench generates the stimuli waveforms for the input signals of the unit under test by either a behavioral description or by reading them from a file. It is also possible to have the output signals from the UUT read, checked for correctness or written to a file by the testbench. Figure 9 illustrates the testbench concept.

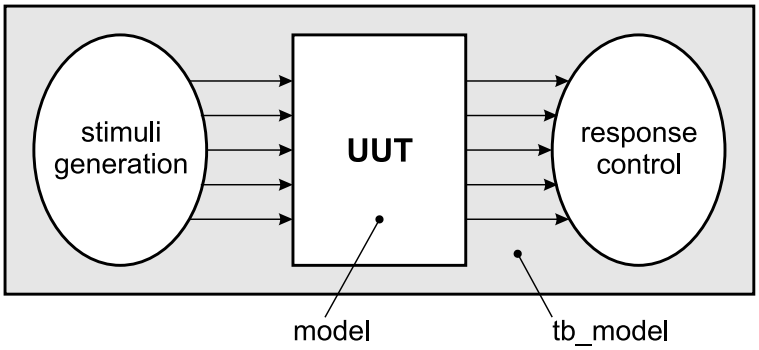


Figure 9: Test environment during simulation

3 Data Types

Every data object, such as a constant, variable or signal, stores a value of the given type: integer, real or bit. The type is specified in the object's declaration. VHDL is a *strongly typed* language. This means that operations and assignments are allowed only if the types of the operands and the result match. In case of mismatch the use of *conversion functions* is required.

Among the basic types of the STANDARD package, which are always recognized by the VHDL, the types of the following two packages are explained:

`std_logic_1164`	in library `IEEE`
`textio`	in library `STD`

3.1 Scalar Types

The basic data types in VHDL are similar to those of programming languages like Pascal or C:

Boolean: Both boolean values `true` and `false` are `boolean` literals.

Integer: Integer numbers in the range $-2^{31}+1$ to $+2^{31}-1$ ($-2\,147\,483\,647$ to $+2\,147\,483\,647$). The default number representation is decimal. In order to use another base it must be explicitly specified:

binary	`2#...#`
octal	`8#...#`
hexadecimal	`16#...#`

Format: [+|-][2#|8#|16#]number[#]

The following additional subtypes of `integer` are defined:

`positive`	: 1 ... n
`natural`	: 0 ... n

Real: Floating point numbers are provided within the range between $-1.0e{+}38$ and $+1.0e+38$. The default number representation is decimal.

Format: [+|-][2#|8#|16#]number.number[#][e[+|-]number]

Character: The `character` literals enumerate the ASCII character set. Non-printing characters are represented by a three-letter name, such as NUL for the null character. Printable characters are represented by themselves, in single quotation marks: `'0'–'9'`, `'a'–'z'`, `'A'–'Z'`. [1]

Bit: The two logical values `'0'` and `'1'` are `bit` literals. [1]

Std_Ulogic and Std_Logic: The standard 1164 of IEEE is defined within the package `std_logic_1164` which resides in the library `IEEE`. It introduces

[1] Due to the strongly typed characteristic of VHDL it can be necessary to specify the type of a value explicitly: `character('1')`
`bit('1')`

a system, named `std_ulogic`, consisting of nine different logical values. The reason for this enhancement is given by the fact, that the type `bit` is not suitable for the precise modelling of digital circuits due to missing values, for example, uninitialized or high impedance states. The type `std_ulogic` consists of the the following elements:

`'U'`	uninitialized
`'X'`	forcing *unknown*
`'0'`	forcing *logical 0*
`'1'`	forcing *logical 1*
`'Z'`	high impedance – for three-state bus systems
`'W'`	weak *unknown*
`'L'`	weak *logical 0*
`'H'`	weak *logical 1*
`'-'`	don't care – for logic synthesis

This data type includes values for modelling three-state bus systems. In addition, a resolution function is defined which allows modeling of multiple drivers for *one* signal (see 7.3, page 53).

Physical Types: A physical type represents a physical value, such as time, length, voltage, and so on. It consists of an integer or a floating point literal followed by a unit name. The package STANDARD defines the type `TIME` with the following units:
`fs, ps, ns, us, ms, sec, min, hr`.
With this physical type delay times can be described as:
`C <= ... after 2 ns;`

Enumeration Types: In order to describe a problem without specifying a coding scheme any appropriate enumeration type can be defined.[2]

Syntax: `type enum_name is (enum_liter {, enum_liter});`

Any desired identifier (first example) or literal (`character` literals in the second example) can be used for `enum_liter`.

Example:
```
-- for traffic lights:
type LIGHT is (RED, YELLOW, GREEN);
-- logic system with four values for simulation:
type FOURVAL is ('X', '0', '1', 'Z');
```

Subtypes: A subtype is a type with a constraint. The constraint defines a subset of values by specifying certain restrictions to the range of the parent type. Such restrictions can also be specified in the declaration of an object. The definition of a subtype has the advantage that it can be done once in a package, and then globally shared.

Example: `subtype DIGIT is integer range 0 to 9;`

[2] The types `character`, `bit` and `boolean` are predefined as enumeration types in the package STANDARD.

```
...
variable MSD, LSD: DIGIT;

— is equal to —
variable MSD, LSD: integer range 0 to 9;
```

3.2 Composite Types

Array: Similarly to other programming languages, an array consists of consecutively numbered elements of the same type. The declaration of an array type includes the name of the array type, the description of the index type and range, and the specification of the element type.

Syntax: `type array_name is array (index_description) of element_type;`

Possibilities for index_description:

index_range |	`integer` range
index_type |	enumeration type as index
index_type `range` index_range |	general description
index_type `range <>`	unconstrained type; range must be specified in variable/signal declaration

Certain characteristics of arrays are explained in the following examples:

Index types: Besides the commonly specified indices of the type integer, user defined enumeration types can also be used.

Example:
```
type INSTRUCTION is
    (ADD, SUB, LDA, LDB, STA, STB, OUTA);
subtype FLAGS is integer range (0 to 7);
...
-- array of flag values
type INSTR_FLAG is array (INSTRUCTION) of FLAGS;
```

Loop varables as index: Indices can be incremented or decremented within a loop by the loop variable.

Example:
```
...
process ...
  type T_SHIFT_MEM is array (0 to 7) of integer;
  variable SHIFT_MEM : T_SHIFT_MEM;
begin
  ...  DATA_OUT <= SHIFT_MEM(7); for I in 7 downto 0 loop
    SHIFT_MEM(I) := SHIFT_MEM(I-1);
  end loop;
  SHIFT_MEM(0) := DATA_IN;
end process;
```

Unconstrained indices: Indices are often declared over the whole range of the index type. In this case the restriction of the range to the desired one must be done in the variable or signal declaration.

Example:
```
-- the declaration within the package STANDARD:
type BIT_VECTOR is array (NATURAL range <>) of BIT;
...
-- now the index range becomes restricted:
variable BYTE: BIT_VECTOR (0 to 7);
```

Index range: The sequence of the index is important:

Example:
```
type AVEC is array (0 to 3) of bit;
type BVEC is array (3 downto 0) of bit;
...
variable AV: AVEC;
variable BV: BVEC;
...
AV := "0101";
                ⇒ AV(0)='0' AV(1)='1' AV(2)='0' AV(3)='1'
BV := "0101";
                ⇒ BV(0)='1' BV(1)='0' BV(2)='1' BV(3)='0'
```

Array assignment: Assignments can be done either by *positional* association, by *named* association or in a mixed way.

Syntax for named association:
```
[typ_name']                          optional type qualifier
( selector => expression{,
  selector => expression}[,
  others => expression] )
```

Example:
```
variable C: BIT_VECTOR (0 to 3);
variable H, I, J, K: bit;

possible assignments
C := "1010";                                   4-bit string
C := H & I & J & K;                          concatenation
C := ('1', '0', '1', '0');                       aggregate

array aggregates
C := ('1', I, '0', J or K);             positional association
C := (0=>'1', 3=>J or K, 1=>I, 2=>'0');   named association
C := ('1', I, others => '0');                        mixed
```

An aggregate is a set of comma-separated elements enclosed in parenthesis.

Slice of an array: It can be choosen by specifying the desired index range.

Example:
```
variable A: BIT_VECTOR (3 downto 0);
variable B: BIT_VECTOR (8 downto 1);
...
B(6 downto 3) := A;
```

Multi-dimensional arrays: They can be declared by specifying more than one index.

Example:
```
type MEMORY is array (0 to 7, 0 to 3) of bit;
...                                          8 × 4 bit array
constant ROM: MEMORY := ( ('0','0','0','0'),
```

```
                                ('0','0','0','1'),
                                ('0','0','1','0'),
                                ('0','0','1','1'),
                                ('0','1','0','0'),
                                ('0','1','0','1'),
                                ('0','1','1','0'),
                                ('0','1','1','1'));
variable DATA_BIT: bit;
...
-- access to one element:
DATA_BIT := ROM (5,3);                                  is '1'
```

It is possible to declare an array type which is an array of an other array type. Note the difference in adressing a two-dimensional array of the above the example and an array of an array like the example below.

Example:

```
type WORD is array (0 to 3) of bit;    a 4-bit storage element
type MEMORY is array (0 to 7) of WORD;      8 × 4-bit array
...
constant ROM: MEMORY := ( ('0','0','0','0'),
                          ('0','0','0','1'),
                          ...
                          ('0','1','1','1'));
variable DATA: WORD;
variable DATA_BIT: bit;
variable ADDR, INDEX: integer;
...
DATA := ROM (ADDR);
DATA_BIT := ROM (ADDR)(INDEX);
```

Array subtypes: They can be defined for existing array types or unconstrained types.

Example: `subtype BYTE is BIT_VECTOR (7 downto 0);`
subtype of the unconstrained type `BIT_VECTOR`

The following array types are predefined in the appropriate packages:

String: Array type of the type `character`.
— in package `STANDARD`

```
type STRING
     is array (POSITIVE range <>) of CHARACTER;
```

Bit_Vector: Array type of the type `bit`
— in package `STANDARD`

```
type BIT_VECTOR is array (NATURAL range <>) of BIT;
```

Std_Logic_Vector: Array type of the type `std_logic`
— in package `STD_LOGIC_1164`

```
type STD_LOGIC_VECTOR
     is array ( NATURAL range <>) of STD_LOGIC;
```

In order to use logic or arithmetic operators with the type STD_LOGIC_VECTOR it is necessary to specify whether the value should be interpreted as *unsigned* or *signed* (see 5 on page 40). Where appropriate, this must be done by type conversion functions (see 2.5.3 on page 20) or by declaring the data object of the adequate type. The following two data types are predefined in the package STD_LOGIC_ARITH or NUMERIC_STD:

```
type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
type SIGNED is array (NATURAL range <>) of STD_LOGIC;
```

Record: Data elements of different types can be collected in one data object by defining a record type. This is useful for abstract data objects. Single elements of a record can be accessed by their name.

Example:

```
type TWO_DIGIT is                          numbers from −99 to +99
  record  SIGN  :  bit;
          MSD   :  integer range 0 to 9;
          LSD   :  integer range 0 to 9;
  end record;
...
process ...
  variable ACNTR, BCNTR: TWO_DIGIT;
begin
  ACNTR.SIGN := '1';                       access to a element
  ACNTR.MSD := 1;
  ACNTR.LSD := ACNTR.MSD;
  ...
  BCNTR := TWO_DIGIT'('0',3,6);        aggregate, type qualified
  ...
end process;
```

3.3 Access Types

Data objects of the type access are pointers to dynamically allocated scalar or complex data objects. They are similar to pointers in other programming languages (C or Pascal). Only a variable can be declared of type access.

Syntax: `type ptr_type_name is access type_name;`

In order to allocate and deallocate memory for an access type variable two operators are defined:

New: This function is used to allocate memory for the object to which a variable of type access is pointing. It is, therefore, used in conjunction with an *assignment* to an access type variable. Initial values for the newly created object can be explicitly specified.

If the access type variable is pointing to an unconstrained type like string then the restriction must be defined within the function call new.

Deallocate: This procedure is provided to free the memory allocated for the object to which an access type variable is pointing.

Example:

```
type CELL;                                          incomplete type
type LINK is access CELL;                                access type
type CELL is                          full type declaration for CELL
  record
    VALUE :  integer;
    NEXTP :  LINK;
  end;
variable HEAD, TEMP : LINK;                          pointer to CELL
...
TEMP := new CELL'(0, null);        new data object with initial values
for I in 1 to 5 loop
  HEAD := new CELL;                                 additional objects
  HEAD.VALUE := I;                            access to record element
  HEAD.NEXTP := TEMP;
  TEMP := HEAD;
end loop;
...
deallocate(TEMP);                                     free the memory

allocate new memory
new CELL;                                                   new object
new CELL'(I, TEMP);                                ... with initial values

... with the required range restriction
new BIT_VECTOR (15 downto 0);                  by specifying an index range
new BIT_VECTOR'("001101110");                   by assigning an initial value
```

3.4 File Types

In the package TEXTIO the data types **text** and **line** are declared. These types and some additional functions provide access to text files similarly to other programming languages. This can be useful during simulation, for example, for reading stimuli data from a text file or for storing the output data of the unit under test (UUT).

Example:

```
use std.textio.all;
  -- read data from file-1 (test.dat)
  -- write data to file-2 (out.dat )
entity COPY4 is                                          without ports
end COPY4;

architecture FIRST of COPY4 is
begin
  process (go)
    -- file with the input data:
    file INSTUFF: text is in "\path\test.dat";
    -- file for the output data:
    file OUTFILE: text is out "\path\out.dat";
    variable L1, L2:  line;
    variable VECT: bit_vector(3 downto 0);

    begin
```

```
            while not (endfile(INSTUFF)) loop        until the end of file
              readline (INSTUFF, L1);                          read one line
              read (L1, VECT);                 copy the input data to VECT
              write (L2, VECT);        copy VECT under a pointer to string
              writeline (OUTFILE, L2);          write one line to OUTFILE
            end loop;
          end process;
      end FIRST;
```

The package TEXTIO consists mainly of the following declarations and definitions:

`type LINE:` A pointer to a string value.

`type TEXT:` A file of variable length with ASCII records.

`file INPUT:` The standard input device.

`file OUTPUT:` The standard output device.

`procedure READLINE:` Input routine to read one line of the input file into the string pointer `line`. The allocation of memory for the string object is done automatically.

`procedure READ:` Copy the contents of the string pointer into an object of one of the predefined types `bit`, `bit_vector`, `boolean`, `character`, `string`, `integer`, `real` or `time`. The memory required for the copied data is freed afterwards.

`procedure WRITE:` Copy or add a data object of the above mentioned types to the contents of a string pointer `line`. Various possibilities exist to specify the width of the string and to justify the data. The allocation of memory is done also automatically.

`procedure WRITELINE:` Write the data to which the string pointer `line` points to the output file. Afterwards, the memory needed to store the data to which `line` points is also freed.

`function ENDFILE:` This function returns a boolean value which indicates whether or not the end of file is reached.

All declarations found in the package TEXTIO are itemized in Appendix A.

3.5 Type and Field Attributes

It is possible to write more general VHDL codes using attributes. The desired properties of objects or types are then determined by the attributes during the elaboration phase. Attributes can be classified into the following categories:

Dimension: These attributes determine ranges and bounds of arrays and enumeration types, or of signals/variables of these types. In the case of multidimensional arrays the index number must be specified.

Syntax: The bounds of a range

`...'left[(n)]`	left bound (of the nth dimension)
`...'right[(n)]`	right bound (of the nth dimension)
`...'high[(n)]`	upper bound (of the nth dimension)
`...'low[(n)]`	lower bound (of the nth dimension)

Length of arrays

`...'length[(n)]`	number of elements (in the nth dimension)

Ranges

`...'range[(n)]`	range `..to/downto..`	(of the nth dimension)
`...'reverse_range[(n)]`	range `..downto/to..`	(of the nth dimension)

Example: The bounds of ranges

```
type T_RAM_DAT is array (0 to 511) of integer;
variable RAM_DAT: T_RAM_DAT;
...
for I in RAM_DAT'low to RAM_DAT'high loop
  ...
```

The bounds of ranges for a multidimensional array

```
variable MEM (0 to 15, 7 downto 0) of MEM_DAT;
...
MEM'left(1)          is 0
MEM'right(1)         is 15
MEM'left(2)          is 7
MEM'right(2)         is 0
MEM'low(2)           is 0
MEM'high(2)          is 7
```

Length of arrays

```
type BIT4 is array (0 to 3) of BIT;
type BIT_STRANGE is array (10 to 30) of BIT;
...
BIT4'length          is 4
BIT_STRANGE'length   is 21
```

Ranges

```
function VEC2INT (INVEC: bit_vector) return integer is
  ...
begin
  for I in INVEC'range loop
    ...
```

Position: These attributes determine values, positions or the base types of enumeration or physical types.

Syntax: Values

`type'succ(value)`	Value of the parameter whose position is one larger than the position of `value`
`type'pred(value)`	Value of the parameter whose position is one less than the position of `value`

`type'leftof(value)`	Value of the parameter that is to the left of `value`
`type'rightof(value)`	Value of the parameter that is to the right of `value`
Position information	
`type'pos(value)`	Position of `value`
`type'val(position)`	Value of `position`
Base type	
`type'base`	Base type to the `subtype type`

Example:

```
type COLOR is (RED, BLUE, GREEN, YELLOW, BROWN, BLACK);
subtype TLCOL is COLOR range RED to GREEN;
...
COLOR'low                 is RED
COLOR'succ(RED)           is BLUE
TLCOL'base'right          is BLACK
COLOR'base'left           is RED
TLCOL'base'succ(GREEN)    is YELLOW
```

Several characteristics of signals can be tested by signal attributes. For example, whether a signal changed its value or has been stable for a certain amount of time can be determined by these attributes.

The predefined attributes for signals are explained in Section 7.4 on page 56.

4 Declarations and Identifiers

Identifiers are used to assign programmer defined names to objects. With the exception of a few reserved words, any word could be used. Following rules apply:

1. Characters `'a'...'z'`, `'0'...'9'`, `'_'`.
2. the first character must be a letter.
3. VHDL is not case sensitive.

When reference to Libraries and Packages is made, the complete object name must be given of the form:
lib_name.package_name.item_name

Comments: start with two adjacent hyphens-- and extend to the end of the line.

Constants: assign a specific value to an object within a `package`, `entity` or `architecture`, and preserve it throughout the entire design.

Syntax: `constant` identifier: type [range_expr][:= expression];

Example:
```
constant Vcc:   real       := 4.5;
constant CYCLE: time       := 100 ns;
constant FIVE:  bit_vector := "0101";
```

Variables: contain values assigned within a `process`. These are used sequentially according to the control flow. Variables *can not* be used to exchange information between different processes.

Syntax: `variable` identifier_list: type [range_expr][:= expression];

Variable declarations may specify the range of the data type and optionally initialize it to the desired value within that range.

Example:
```
variable INDEX:     integer range 1 to 60 := 27;
variable CYCLE_TIME:time range 10 ns to 50 ns := 10 ns;
variable REGISTER:  std_logic_vector (7 downto 0);
```

Signals: connect together *Design-Entities* and propagate value changes within a design. They are the primary means of communication between processes.[3]

Syntax: `signal` identifier_list: type [range_expr][:= expression];

Signal declaration may specify the range of the data type and optionally initialize it to the desired value within that range.

[3]Due to the importance of signals in VHDL they are given a more detailed description in Section 7.

Example:
```
signal COUNT:     integer range 1 to 31;
signal GROUND:    bit := '0';
signal INT_BUS:   std_logic_vector (1 to 8);
```

Caution:
Signals *can not* be declared within a process. They can be used within a `process`; however, signal value assignment occurs in the simulation time. That is, signal values are *not* updated in the sequential order, as is the case with variables, rather at the `wait`-Statement. At this point the signal acquires a value assigned to it immediately before the `wait`-Statement.

Signal assignments are using special operators to indicate their peculiar time behavior. Explicitly stated signal assignment delays take effect during the simulation:

```
signal xyz:  bit;
...
xyz <= '1' after 5 ns;
```

The use of signals in the sequential flow of processes often produces unanticipated erroneous results. Therefore, it is recommended to use *variables* in the sequential flow of a process (with *read and write operations*) and then to assign newly computed values to signals just before the next `wait` statement.

Two additional remarks regarding the variables and signals:

Initialization: variables and signals that are not explicitly initialized during the declaration receive default values according to the following rules:

Enumerated types : the first values in the list
`integer`, `real` : the lowest allowable value

The initialization of enumerated types is often used, for example, by defining the desired initial state of finite state machines as the first one in the list of states, or with `std_logic` where the special value `'U'` (*uninitialized*) is assigned to every signal/variable without an explicit initial value.

it may be desirable to start with `'U'` (*uninitialized*) value.

Range constraints: in order to obtain correct hardware synthesis results, variables and signals must be constrained to the desired bit-width. This is especially critical with the unspecified `integer`-type synthesis, where a default 32-bit datapath is generated.

Example:
```
signal CNT100:    integer range 0 to 99;      unsigned 7-bit
signal ADDR_BUS:  std_logic_vector (7 to 0);          8-bit
```

5 Expressions and Operators

Expressions in VHDL may be constructed using the operators listed in the table in order of increasing precedence. The desired precedence may also be achieved through the explicit use of parentheses.

Operator	Function	Operands Type1	- Type2
Logical Operators			
`and`	$a \wedge b$	bit, bit_vector, boolean	- =
`or`	$a \vee b$	bit, bit_vector, boolean	- =
`nand`	$\neg(a \wedge b)$	bit, bit_vector, boolean	- =
`nor`	$\neg(a \vee b)$	bit, bit_vector, boolean	- =
`xor`	$a \neq b$	bit, bit_vector, boolean	- =
Relational Operators			
`=`	$a = b$	same type	
`/=`	$a \neq b$	same type	
`<`	$a < b$	same type	
`<=`	$a \leq b$	same type	
`>`	$a > b$	same type	
`>=`	$a \geq b$	same type	
Arithmetic Operators - Additive			
`+`	$a + b$	integer, real	- =
`-`	$a - b$	integer, real	- =
`&`	$a\&b$	bit, bit_vector, character, string	- same type
Arithmetic Operators - Sign			
`+`	$+a$	integer, real	
`-`	$-a$	integer, real	
Arithmetic Operators - Multiplicative			
`*`	$a * b$	integer, real	- =
`/`	a/b	integer, real	- =
`mod`	a div b	integer	- =
`rem`	a mod b	integer	- =
Other Operators			
`**`	a^b	integer, real	- integer
`abs`	$\|a\|$	integer, real	
`not`	$\neg a$	bit, bit_vector, boolean	

Since VHDL is a strongly typed language, it is sometimes useful to perform conversions between different types as well as explicitly specify the exact type that the expression should attain.

Qualified Expressions: allow explicit specification of the type. This is helpful when there is no unambiguous classification of objects.

Syntax: `type'(expression)`

Example:
```
type MONTH is (APRIL, MAY, JUNE);
type NAMES is (APRIL, JUNE, JUDY);

...  MONTH'(JUNE) ...                    for months
...  NAMES'(JUNE) ...                    for names
```

Conversions: are used to convert an object of one type to another. Conversion functions for the standard types are predefined. A user is responsible for conversions between user-defined types.[4]

Example:
```
type FOURVAL is ('X', 'L,', 'H', 'Z');      four-value logic
type VALUE4 is  ('X', '0,', '1', 'Z');    ..., different logic
...
function CONVERT4VAL (S: FOURVAL) return VALUE4 is
begin                                     conversion function
  case S is
    when 'X' => return 'X';
    when 'L' => return '0';
    when 'H' => return '1';
    when 'Z' => return 'Z';
  end case;
end CONVERT4VAL;
...
process (ABC)                   ...calls the conversion function
  variable ABC: FOURVAL;
  variable XYZ: VALUE4;
...
  XYZ := CONVERT4VAL (ABC);
...
```

IEEE 1164 – Std_Logic_Vector

Special operators are defined for the `std_(u)logic_vector` data type. In order to distinguish between *unsigned* and *signed* (2's complement) binary representations either explicit type conversion are required (see Section 2.5.3) or one of the two packages `std_logic_unsigned` or `std_logic_signed` must be used. These two packages as well as the basic package `std_logic_1164` are defined in the `IEEE` library. They are especially useful for the evaluation of comparison operations. These packages are:

[4]Functions are described in Section 6.2, page 47.

std_logic_1164

logic	`and nand or nor xnor not`

std_logic_unsigned / std_logic_signed

relational	`= /= < <= > >=`
arithmetic	`+ -, + - abs, *`

In addition, these packages contain a number of conversion functions as well as shift operations for vectors.

Example:

```
library IEEE;                                   specify the Library
use IEEE.STD_LOGIC_1164.ALL;                     specify packages
use IEEE.STD_LOGIC_SIGNED.ALL;
...
  VARA := "1011";                                        = -5
  VARB := "0011";                                         = 3
  if (VARA > VARB) then                                 false
  ...
```

If the unambiguous classification of an object with respect to the number system is not possible, as is the case with literal characters, subtypes (**unsigned**, **signed**) can be explicitly given.

Example: `signed'("1011") > signed'("0011")` `false`

6 Sequential Modeling

A process plays the central role in sequential VHDL descriptions. The `process`-Statement is used for behavioral descriptions of architectures. It defines code segments where internally all statements are processed in sequence, one after another.

`Process`-Statements behave like concurrent statements with respect to the rest of a design. At any given time there may be many different processes active, and their order of execution in VHDL code is irrelevant.[5]

Syntax:

```
[proc_label:] process [(sensitivity_list)]
  [subprogram_decl|subprogram_body]
  [type_decl]
  [subtype_decl]
  [constant_decl]
  [variable_decl]
  [file_decl]
  [alias_decl]
  [attribute_decl]
  [attribute_spec]
  [use_clause]
begin
  [sequential_statements]
end process [proc_label];
```

Note that the optional label (proc_label) is extremely useful for debugging purposes during the simulation, and therefore should not be omitted.

The example below demonstrates the use of two processes to determine the maximum and minimum values which appear on the input ports.

Example:

```
entity LOW_HIGH is
  port (  A, B, C: in integer;                    Inputs
          MI, MA: out integer);                   Outputs
end LOW_HIGH;

architecture BEHAV of LOW_HIGH is
begin
  L: process                                      find minimum
    variable LOW: integer := 0;
    begin
      wait on A, B, C;
      if A < B    then LOW := A;
                  else LOW := B;
      end if;
      if C < LOW  then LOW := C;
      end if;
      MI <= LOW after 1 ns;
    end process;
```

[5] Additional information on `process`-statements and their execution during the simulation is found in Section 8, page 57.

```
  H: process                                        find maximum
    variable HIGH: integer := 0;
    begin
      wait on A, B, C;
      if A > B    then HIGH := A;
                  else HIGH := B;
      end if;
      if C > HIGH then HIGH := C;
      end if;
      MA <= HIGH after 1 ns;
    end process;
end BEHAV;
```

6.1 Assignments

Signal Assignments: Signal assignments within processes are covered in detail in section 7.2, page 52. Although signal assignments may appear in the sequential order within a process, the way they are handled during the simulation does not necessarily follow that order.

Variable Assignment: changes the value of a variable. Both variable and expression types must be compatible, otherwise conversion functions or attributes must be employed.

Syntax: variable := expression;

If: is a flow control statement for conditional execution of sequential statements. In the hardware sense it represents simple decoding.

Syntax:
```
if condition then
  sequential_statements
{elsif condition then
  sequential_statements}
[else
  sequential_statements]
end if;
```

Case: executes one of several sequences of statements depending on the value of expression. In the hardware sense it means decoding of complex codes, for example, buses.

Syntax:
```
case expression is
  {when choices => sequential_statements}
end case;
```

choices must be of the following types:

value => sequential_statements	exactly one value
value1 | value2 ... => sequential_statements	
	Range of values
value1 to value2 => sequential_statements	chosen range
others => sequential_statements	remaining range

All possible values of expression must be covered by the set of choices. Alternatively, `when others` could be used as a last statement matching the remaining (uncovered) choices.

Example:
```
case BCD is                                   Decoder: BCD to 7-Segment
  when "0000" => LED := "1111110";
  when "0001" => LED := "1100000";
  when "0010" => LED := "1011011";
  when "0011" => LED := "1110011";
  when "0100" => LED := "1100101";
  when "0101" => LED := "0110111";
  when "0110" => LED := "0111111";
  when "0111" => LED := "1100010";
  when "1000" => LED := "1111111";
  when "1001" => LED := "1110111";
  when others => LED := "-------";                    don't care
end case;
```

Loop: allows repeated execution of a sequence of statements; represents repetition of elements (for example, corresponding to the bit-width) in hardware.

Syntax:
```
[loop_label:] while condition loop |                  with condition test
[loop_label:] for identifier in value1 to value2 loop |        counter
[loop_label:] loop                                     forever loop...
  sequential_statements
end loop [loop_label];
```

The identifier of the `for`-loop doesn't need to be declared anywhere; it serves as a local variable of the loop. Value assignments to the identifier are not possible.

Next: statement forces the current loop iteration to terminate. Additional conditions associated with the termination may be specified.

Syntax: `next [loop_label][when condition];`

Example:
```
for I in 0 to MAX_LIM loop
  if (DONE(I) = true) then next;          Jump to end loop
  end if;
  Q(I) <= A(I);
end loop;

L1:  while J < 10 loop                                  outer loop
  L2:  while K < 20 loop                                inner loop
    ...
    next L1 when J = K;                 jump out of the inner loop
    ...
  end loop L2;
end loop L1;                                         jump destination
```

Exit: terminates a loop; optional conditions may be specified.

Syntax: `exit` [loop_label][`when` condition];

Example:
```
for I in 0 to MAX_LIM loop
  if (Q(I) <= 0) then exit;          jump out of the loop
  end if;
  Q(I) <= (A * I);
end loop;
...                                  jump destination
```

Assert: allows to check whether certain conditions are satisfied during the program execution under the VHDL-Simulator. This option is helpful for double-checking time restrictions (set-up, hold ...), range constraints, and so on.

Syntax:
```
assert condition
  [report string_expr]
  [severity failure|error|warning|note];
```

If a condition is evaluated to the boolean *false*, the user-specified string_expr is displayed along with the severity level indicator.

Example:
```
process (CLK, DIN)                   behavior of a D-FF
  variable X: integer;
  ...
begin
  ...
  assert (X > 3)
    report "setup violation"
    severity warning;
  ...
end process;
```

Wait: dynamically controls execution and suspension of processes. At the behavior level it makes possible to give a realistic representation of a process by modeling signal-dependent activities. Furthermore, through the `wait` statement signal values are updated in a circuit[6].

Syntax:
```
wait
  [on signal_name {, signal_name}]
  [until condition]
  [for time_expr];
```

A sensitivity_list of a `process` is functionally equivalent to the `wait on ...` appearing at the end of the process. There are four different ways to use the `wait`-statement:

[6] Section 7.2 (page 52) explains the way `wait`-statements work in greater detail.

- `wait on A, B;` : suspends a process until an occurrence of a change is registered. As shown here, execution will resume when a new event is detected on either signal `A` or `B`.
- `wait until X > 10;` : suspends a process until the condition is satisfied; in this case, until the value of a signal `X` is `> 10`.
- `wait for 10 ns;` : suspends a process for the time specified; here, until `10 ns` of simulation time elapses.
- `wait;` : suspends a process indefinitely... Since a VHDL-`process` is always *active*, this statement at the end of a process is the only way to suspend it. This technique may be used to execute initialization processes only once.

The example below models an architecture which simulates a Producer/Consumer problem using two processes. The processes are synchronized through a simple handshake protocol, which has two wires, each with two active states.

Example:

```
entity PRO_CON is
  ...
end PRO_CON;

architecture BEHAV of PRO_CON is
  signal PROD: boolean := false;  item produces a semaphore
  signal CONS: boolean := true;   item consumes a semaphore
begin
  PRODUCER: process                                  producer model
    begin
      PROD <= false;
      wait until CONS;
      ..produce item
      PROD <= true;
      wait until not CONS;
    end process;

  CONSUMER: process                                  consumer model
    begin
      CONS <= true;
      wait until PROD;
      CONS <= false;
      ..consume item
      wait until not PROD;
    end process;
end BEHAV;
```

6.2 Subprograms

In VHDL, procedures (one or more return parameters) and functions (only one return value) are available as subprograms. Functions are often used for type conversions or as resolution functions (see Section 7.3, page 53).

Local variables can be declared within subprograms and their values are defined only until the return from a subprogram occurs. In contrast, variables of a `process` correspond to the local memory locations.

A subprogram must be declared prior to its call. Therefore, if it is called within a `process`, it must be declared in its respective `architecture`, `entity` or `package`.

During the subprogram call, passing of parameters is done by either the proper position in the parameter list or by the name, such that declaration_name => actual_parameter.

Function: may have several parameters but produces only one return value (analogous to the VHDL expression).

Syntax:
```
function func_name (parameter_list)
return type_name is
  [variable_declaration]
  [constant_declaration]
  [type_declaration]
  [use_clause]
begin
  [sequential_statements]
  return expression;
  [sequential_statements]
end [func_name];
```

In the following example, function VEC2INT converts a bit-vector into an integer value.

Example:
```
architecture ...
  ...
  function VEC2INT (S: bit_vector range 1 to 8)
  return integer is
    variable RES: integer := 0;                 local variable
  begin
    for I in 1 to 8 loop
      RES := RES * 2;
      if S(I) = '1'  then RES := RES + 1;
      end if;
    end loop;
    return RES;
  end VEC2INT;
  ...
begin
  ...
  process ...
    ...
    XVAL := VEC2INT (XBUS);                     function call
```

```
    ...
  end process;
  ...
end ...
```

Procedure: can have zero or more parameters with the following modes:

- `in` : readable only within the procedure
- `out` : writable only; the use of these parameter is allowed *only* on the left side of an assignment.
- `inout` : read/write paramenter, which can be universally used within a procedure.

Procedure parameters can be both variables and signals (after their explicit declaration). In VHDL code procedures are handled similarly to the assignments.

Syntax:

```
procedure proc_name (parameter_list) is
  [variable_declaration]
  [constant_declaration]
  [type_declaration]
  [use_clause]
begin
  sequential_statements
end [proc_name];

parameter_list:
[variable]     name_list [in|out|inout] type_name
                                        [:= expression]; |
signal         name_list [in|out|inout] type_name;
```

The example below shows a procedure which performs the same bit-vector conversion into an integer value as the function of the previous example. In addition, the procedure also sets a Flag.

Example:

```
architecture ...
  ...
  procedure VEC2INT                          Declaration
    ( S: in bit_vector;
      ZFLAG: out boolean;
      Q: inout integer ) is                  mode assignments
  begin
    Q := 0;
    ZFLAG := true;
    for I in 1 to 8 loop
      Q := Q * 2;                   Q allowed on the right side
      if S(I) = '1' then
        Q := Q + 1;                      ⇒ Mode is:  inout
        ZFLAG := false;
      end if;
    end loop;
  end VEC2INT;
```

```
begin   -- architecture statement part
  ...
  process ...
    ...
    VEC2INT (XBUS, XFLG, XVAL);                procedure call
    ...
  end process;
  ...
end ...
```

Overloading
VHDL, similarly to other programming languages, allows Overloading of prodecures and functions. *Overloading* means that two or more subprograms have the same name but differ in the number of parameters and base types. When an overloaded subprogram is called its name, number of actual parameters, order of arguments and their types, are used to determine which function/procedure should be invoked. Overloading overcomes strong typing and allows a more general use of operators and functions.

Argument-Type: overloaded subprograms are distiguished through the different types of their arguments (parameters).

Example:
```
function DECR (X: integer) return integer is
begin
  ...
end DECR;

function DECR (X: real) return real is
begin
  ...
end DECR;
...
variable A, B: integer;
...
B := DECR(A);                    call the first (integer) function
```

Argument-Number: overloaded subprograms are distinguished based on the different number of parameters.

Example:
```
function CONV_ADDR (A0, A1:  bit) return integer is
begin                                              2 arguments
  ...
end;
function CONV_ADDR (A0, A1, A2:  bit) return integer is
begin                                              3 arguments
  ...
end;
```

In general, overloading permits extension of already existing operators found in the default package STANDARD. This is particularly useful for writing vendor/

user-specific packages. Some of the most commonly used extended packages are n-value logic packages MVL7, MVL9 and STD_LOGIC_1164. They define logic values 'X' (unknown), 'Z' (high-impedance), etc., as well as drivers of different strengths, in addition to the conventional '0' und '1'. Logical (and, or, not, xor...), arithmetic (+, −, ∗...) and comparison operators (=, / =, >, <...) for the new types are also provided. For the often used STD_LOGIC_1164 package an extension exists which defines operators for unsigned and signed (2's complement) binary representations. The user can then take advantage of overloading and conveniently use these types with their respective operators. For functions with only two arguments it is possible to use the *infix-Notation*.

In the following example, a new addition function for a 4-bit `bit_vector` is defined.[7]

Example:
```
function "+" (A, B: bit_vector (3 downto 0))
                                          return bit_vector is
  variable SUM: bit_vector (3 downto 0);
  variable CARRY: bit;
begin
  CARRY := '0';
  for I in 0 to 3 loop
    SUM(I)  := A(I) xor B(I) xor CARRY;
    CARRY   := ((A(I) and B(I)) or (A(I) and CARRY)
                                        or (B(I) and CARRY));
  end loop;
  return SUM;
end;
```

[7]There is no standard `bit_vector` addition in VHDL!

7 Signals

VHDL elements at the behavior level (processes, variables and sequential assignments) find their naturally corresponding elements in programming languages. Signals and parallel blocks, on the other hand, have characteristics that are typical for structure-level descriptions and their simulations.

In VHDL signals are the only way to tie together elements of structural descriptions or to enable processes to communicate. They act like *wires*. In the simulation, signals are handled in a time dependent manner, in the sense that every action is preceded by a cause.

7.1 Signal Declaration

In VHDL code signals may be declared:

1. within a `package` for global signals.
2. as `ports ...` in the `entity` declaration for entity-global signals.
3. within an `architecture` as architecture-global signals.

Syntax: in `package` or `architecture`

```
signal signal_name:  type_name [:= expression];
```

as a `port` of an `entity`

```
...  signal_name:  in|out|inout|buffer type_name;
```

When ports of an `entity` are declared, direction of the information flow must be specified:

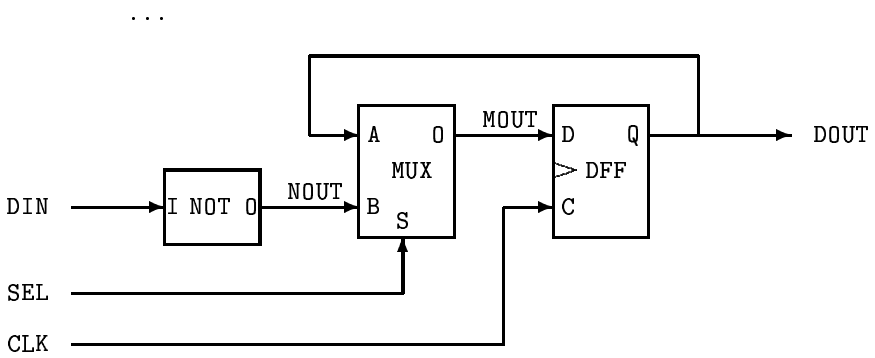
`in`	input — read port; such signals may appear only on the right side of variable/signal assignment.
`out`	output — write port; must be used on the left hand side of a signal assignment.
`inout`	bidirectional wire; no restriction in the usage.
`buffer`	generally, it is an output port (driver); however, it may be used on the right hand side of assignments.

Example:

```
package SIGDECL is
  signal VCC: std_logic := '1';                    global signals
  signal GND: std_logic := '0';
end SIGDECL;

entity MUXFF is                                  entity-global signals
  port (  DIN, SEL, CLK: in bit;
          DOUT: buffer bit);                       used internally
  signal NOUT: bit;
end MUXFF;
...

architecture STRUCT of MUXFF is
  signal MOUT: bit;                        architecture-global signal
begin
```



7.2 Signal Assignments in Process

In general, processes can communicate with the *outside world* (other processes, entities...) only through signals, and signal values (for example, output ports) are assigned in processes. There are, however, important factors to keep in mind.

Delay time: The goal of VHDL description is to simulate *real* circuits with their respective delays (time constants of electric circuits). These delay values are specified during signal assignment statements. For the circuit simulation this means that new signal values are updated only after the specified delay time elapses.

Syntax: signal_name <= expression [after time_expr {, expression after time_expr}];

The delay time specified in the signal assignemt (`after ...`) is considered relative to the simulation time reached before the assignment. Zero delay time is also allowed. In a single signal assignment several delays may be given. The simulation algorithm then arranges the time sequence of the future events in a list (*scheduling*).

Example:
```
R   <= "1010";
S   <= '1' after 4 ns, '0' after 7 ns;
T   <= 1 after 1 ns, 3 after 2 ns, 6 after 8 ns;
CLK <= not CLK after 50 ns;
```

Activation of Assignment: Although signal assignments within a process are surrounded by statements which are processed in sequential order, they *are not activated in the same apparent sequential order.* Signal assignments are activated once the `wait` statement of a process is reached. Alternatively, when a process is used with the *sensitivity-list*, signal assignments occur at the end of the process. This leads to the following consequences:

1. Within a process signals can not be used as variables for temporary value storage.

2. There should be only one assignment per signal in a process; thus indicating *a single* driver.

Due to the peculiarities associated with signal assignments, erroneous circuit behavior is often produced (especially by the VHDL beginners). Therefore, here is one more example on the topic:

Example:

```
X <= Y;   both assignments will be processed at the wait statement
Y <= X;                 ⇒ the values of X and Y will be exchanged
wait ...        ⇒ the sequence of these two statements is irrelevant

V := 1;                                  V becomes 1 — immeditately
S <= V;                 S will be V (also 1) — at the wait statement
A := S;                   A receives the old value of S — immediately
wait ...

X <= 1;     Caution: will be ignored due to the second assignment!
Y <= 3;                         Y will be 3 — at the wait statement
X <= 2;       this assignment overwrites the first assignment above:
wait ...                  ⇒ X will be 2 — at the wait statement
```

7.3 Implicit Type Resolution and Drivers

All previously discussed signal assignments were assumed to have a single driver for each signal. In order to describe such hardware structures as (bidirectional) buses, wired-or, wire-and, and so on, special mechanisms, known as *resolution functions*, are used.

In general, VHDL was not designed to handle signals with multiple drivers. The reason is that either wired-or, wired-and or some other logical behavior is produced depending on the employed circuit implementation methodology. Therefore, it was assumed that a vendor is responsible to deliever required packages with multi-valued logic type declarations and corresponding resolution functions.

There are, however, methods to deal with situations when several drivers are driving a signal. To have multiple drivers usually means that a signal is assigned concurrently by parallel processes, concurrent signal assignments, concurrent procedure calls, and so on.

Types and Subtypes: A `subtype`, which has an associated resolution function, is declared for the specified data type.

Syntax: `subtype subtype_name is resolution_func_name type_name;`

With this approach, when a signal assignment of type `subtype_name` happens, an associated function, which determines the final signal value, is *implicitly* called.

Analogous to the `subtype`, a resolved signal can also be declared as: `signal_name : resolution_func_name type_name;`

Resolution Functions: A *resolution function* is declared as a conventional function and has following characteristics:

as usual:
- arguments are of the *input* mode and are *passed by value.*
- a function returns only one value.

additional features of resulution functions:
- an array of variable length with elements of the type type_name is passed as the function argument.
- the return value of the function is of the *original* type: type_name.
- the function is associated with a **subtype**; the function is called every time a signal of this type is assigned.

In the following example, a *wired-or* resolution function with a four-value data type is described. A tristate driver is modeled by two processes which write to the same output signal.

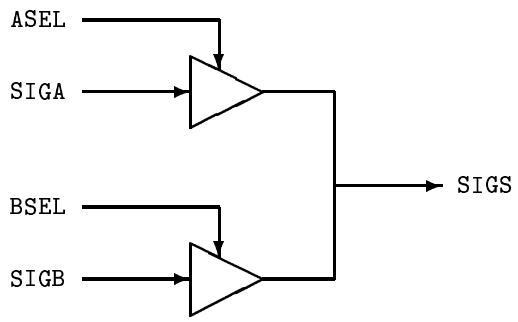
Example:

```
— 4-value type and the corresponding array type for the function —
type BIT4 is ('X', '0', '1', 'Z');
type BIT4_VECTOR is array (integer range <>) of BIT4;

— resulution function —
function WIRED_OR (INP: BIT4_VECTOR) return BIT4 is
  variable RESULT: BIT4 := '0';          result, bus with a pull-down
begin
  for I in INP'range loop                              for each input
    if INP(I) = '1' then
      RESULT := '1';
      exit;                                       jump out of the loop
    elsif INP(I) = 'X' then
      RESULT := 'X';
    else null;                              INP(I) = 'Z' or = '0'
    end if;
  end loop;
  return RESULT;
end WIRED_OR;

— subtype with the resolution function —
subtype RESOLVED_BIT4 is WIRED_OR BIT4;
...
architecture BEHAVE of TRISTATE is
  signal ASEL, BSEL: boolean;                          select signals
  signal SIGA, SIGB: BIT4;                                     inputs
  signal SIGS: RESOLVED_BIT4;                     resolved output signal
begin
  SOURCE1:  process (ASEL, SIGA)                     first output source
  begin
    SIGS <= 'Z';
    if (ASEL) then
      SIGS <= SIGA;
    end if;
  end process;
```

```
  SOURCE2:  process (BSEL, SIGB)              second output source
  begin
    SIGS <= 'Z';
    if (BSEL) then
      SIGS <= SIGB;
    end if;
  end process;
end BEHAVE;
```



The STD_LOGIC_1164 package defines resolution functions for std_logic data types. Types std_logic / std_logic_vector, which implicitly have resolution functions, are subtypes of std_ulogic / std_ulogic_vector (**u**nresolved).

The following example illustrates modeling of bidirectional buses. Here is the description of a 4-bit bus driver/receiver.

Example:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity BUSIO is
  port( OEN:    in std_logic;
        IBUS:   in std_logic_vector (3 downto 0);
        OBUS:   out std_logic_vector (3 downto 0);
        IOBUS:  inout std_logic_vector (3 downto 0));
end BUSIO;

architecture BEHAV of BUSIO is
  begin
    P: process (OEN, IBUS, IOBUS)
    begin
      if (OEN = '1') then                          drive bus
        IOBUS <= IBUS;
      else                                         read bus
        IOBUS <= "ZZZZ";              explicit assignment of 'Z'
      end if;
      OBUS <= IOBUS;
    end process;
  end BEHAV;
```

7.4 Signal Attributes

Besides attributes associated with types, VHDL also provides attributes that are related to signals. Using these attributes it is possible to produce VHDL descriptions that take into consideration dynamic signal behavior. These attributes allow to incorporate simulation events and time instances at the time of simulation execution.

Syntax:

current point in time	
`signal'event`	`true`, if *event* (signal transition)
`signal'active`	`true`, if *transaction* (signal assignment)
past events	
`signal'last_event`	elapsed time from the last signal change
`signal'last_value`	previous signal value
`signal'last_active`	elapsed time from the last signal assignment

Example:

```
entity DFLOP is                                          D-Type FF
  port (  CLK, D: in std_logic;
          Q: out std_logic)
end DFLOP;
architecture BEHAV of DFLOP is
begin
  process (CLK)
  begin
    if  (CLK = '1') and                                  CLK = 1
        (CLK'event) and                              and a new event
        (CLK'last_value = '0')
                          and previous value was 0 (because of 'X'...)
      then  Q <= D;                                  ⇒ rising edge
    end if;
  end process;
end BEHAV;
```

8 Concurrent Modeling

Concurrent statements serve to model the behavior of hardware components where events often occur simultaneously.

Process: A `process` as a whole is treated as a concurrent assignment. It was already presented as a module which contains a set of sequentially executed statements.[8] It has the following features:

- all processes are active in *parallel.*
- a process defines a region in the code where statements are executed sequentially (similarly to the conventional programming languages). It describes behavior employing sequential algorithms.
- a process must contain either a sensitivity-list or explicit `wait` statements.
- within a process, signals belonging to an `entity` or `architecture` could be read and assigned new values.

Process Execution

Since processes are supposed to model the behavior of hardware elements which in real-world are always active, VHDL processes possess some special features.

Execution: A process can be viewed as a forever loop. At the beginning of the simulation, as some sort of initialization, every process is activated and executed up to the first `wait`. Subsequently, the execution of processes suspends according to the conditions enforced by the `wait` statements.

Processes whose conditions of `wait` statements are satisfied, are reactivated. They continue the execution of statements in sequential order until the next `wait` statement suspends them. If the end of a process is reached (`end process;`), the *execution continues from the beginning of a process.* This is illustrated by the following example:

Example:

```
process ...
begin
  loop                                  beginning of the loop
    ...
    wait ...                 at least one wait, or a sensitivity-list
    ...
  end loop;                             end of the loop
end process;
```

Activation: As mentioned above, the simulator executes a process sequentially, statement by statement. It is then suspended at one or more locations by the `wait`s. The execution is again re-activated by the arrival of specific *events.*

[8] See section 6, page 42

It follows then, that a process must have at least one `wait` statement, or a process must be declared with a *sensitivity-list*. The *sensitivity-list* is functionally equivalent to a `wait on ...` statement appearing at the end of a process.

Example:

```
process (A, B)                    sensitivity-list
begin
  ...
end process;

— is equivalent to —
process
begin
  ...
  wait on A, B;
end process;
```

When describing a data flow, each transaction would correspond to a process, which contains only one statement inside. There is a simpler way to model this by using concurrent assignments. These assignments are located within an architecture and each of them corresponds to a process. The order of concurrent assignments in VHDL is irrelevant.

Concurrent Signal Assignment: is functionally equivalent to a process which contains only one statement and has a sensitivity-list.

Syntax: [label:] signal_name <= expression [after time_expr];

The assignment is activated when at least one signal on the right side of the assignment statement changes.

Example:

```
architecture VER1 of MUX is
begin
  OUTPUT <= A (INDEX);
end VER1;

— is equivalent to —
architecture VER1 of MUX is
begin
  process (A, INDEX)
  begin
    OUTPUT <= A (INDEX);
  end process;
end VER1;
```

Selected Signal Assignment: corresponds to a process with a single signal assignment which is enabled through the `case` statement.

Syntax:

```
[label:] with expression select
  signal_name <=  expression when value{,
                  expression when value};
```

The selected signal assignment is activated as soon as one of the signals belonging to the selection condition or expression changes.

Example:
```
with MYSEL select
  Z <=  A when 15,
        B when 22,
        C when 28;
```

Concurrent Procedure Call: is equivalent to a process which contains only one statement — a procedure call. The procedures' parameters are of one of the following modes: `in`, `out` or `inout`. When one of the parameter signals changes, the procedure call is activated.

Example:
```
architecture ...
  procedure VEC2INT
    ( signal S: in bit_vector;
      signal ZFLAG: out boolean;
      signal Q: inout integer;) is
  ...
begin
  VEC2INT (BITVEC, FLAG, NUMBER);
  ...
```

— is equivalent to —
```
architecture ...
  procedure VEC2INT ...            declaration, same as above
begin
  process (BITVEC, NUMBER)
  begin
    VEC2INT (BITVEC, FLAG, NUMBER);
  end process;
  ...
```

Block: In order to efficiently group concurrent assignments, `block` statements may be used. A block may contain declarations of data types, signals, and so on, all of which are locally used. The body of the block statement contains any of the concurrent statements mentioned previously.

A guarded block contains an additional boolean expression guard_expression, which drives an *implicit* signal `GUARD` of `boolean` type. This signal can be used within a block for the control of concurrent assignments. If concurrent statements have an associated `GUARD` signal, they are known as *Guarded Signal Assignments.*

Syntax:
```
label:  block [(guard_expression)]
  [use_clause]
  [subprogram_decl subprogram_body]
  [type_decl]
  [subtype_decl]
  [constant_decl]
  [signal_decl]
  [component_decl]
```

```
begin
  [concurrent_statements]
end block [label];
```

Guarded Signal Assignment: is a special form of the concurrent assignment. The assignment is activated after the `GUARD` signal, which must be of the `boolean` type, is evaluated to `true`. The `GUARD` signal can be explicitly declared and used; however, it is more common to use it implicitly within a *Guarded Block*.

Syntax: `[label:] signal_name <= guarded expression [after time_expr];`

Conditional Signal Assignment: is equivalent to a process with a single `if` statement and a signal assignment. Building complex `if ... elsif ...` conditions is also allowed.

Syntax:
```
[label:] signal_name <= expression when condition else
                        {expression when condition else}
                        expression;
```

The conditional signal assignment is activated as soon as one of the signals belonging to the condition or expression changes.

Example:
```
Z <=  A when (X > 3) else
      B when (X < 3) else
      C;
```

9 Structural Descriptions

A description style where different components of an architecture and their interconnections are specified is known as a VHDL structural description. Initially, these components are declared and then components' instances are generated or instantiated. At the same time, signals are mapped to the components' ports in order to connect them like wires in hardware. VHDL simulator handles component instantiations as concurrent assignments.

Syntax: component declaration:

```
component component_name
  [generic ( generic_list:  type_name [:= expression] {;
             generic_list:  type_name [:= expression]} );]
  [port ( signal_list:  in|out|inout|buffer type_name {;
          signal_list:  in|out|inout|buffer type_name} );]
end component;
```

component instantiation:

```
component_label:  component_name port map (signal_mapping);
```

The mapping of ports to the connecting signals during the instantiation can be done through the positional notation. Alternatively, it may be done by using the named notation, using the already familiar format:

Syntax: signal_mapping: declaration_name => signal_name.

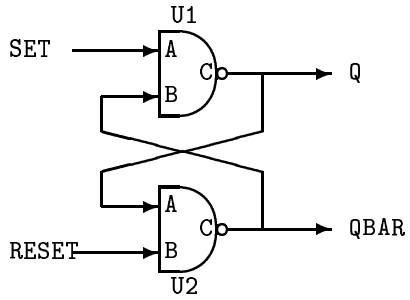
If one of the ports has no signal connected to it (this happens, for example, when there are unused outputs), a reserved word open may be used. A function call can replace the signal name. This allows direct type conversions at the component instantiation.

Example:

```
entity RSFF is
port (  SET, RESET: in bit;
        Q, QBAR: inout bit);
end RSFF;

architecture NETLIST of RSFF is
  component NAND2
    port (A, B: in bit; C: out bit);
  end component;
begin
  U1:  NAND2 port map (SET, QBAR, Q);
  U2:  NAND2 port map (Q, RESET, QBAR);
end NETLIST;

— named notation instantiation: —
  U1:  NAND2 port map (A => SET, C => Q, B => QBAR);
```



9.1 Generation of Instances

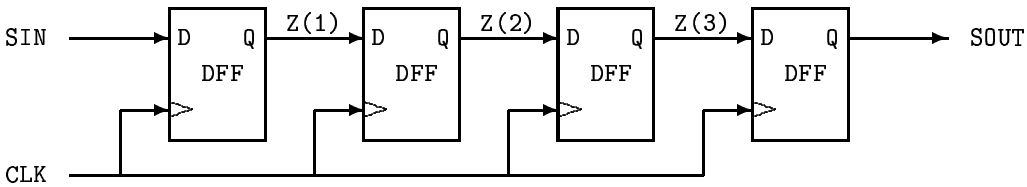
Some repetitive structure descriptions, such as elements corresponding to the bus width, memory size, etc., benefit from the array-like arrangement of components. Descriptions of this type may be done with the `generate` statements, which allow:

1. repetition of structures corresponding to the `for ... loop`.
2. selection of specific instantiations through the `if ... then` conditions.

Syntax:

```
generate_label:  for variable in range generate
  concurrent_statement                                  general instantiations
end generate [generate_label];

generate_label:  if (condition) generate
  concurrent_statement
end generate [generate_label];
```



Example:

```
entity SHIFT is
port (  SIN, CLK: in bit;
        SOUT: out bit);
end SHIFT;

— iterative construction of a shift register —
architecture NETLIST1 of SHIFT is
  component DFF
    port (D, CLK: in bit; Q: out bit);
  end component;
  signal Z: bit_vector (0 to 4);
begin
  Z(0) <= SIN;
  GF: for I in 0 to 3 generate
    UI: DFF port map (Z(I), CLK, Z(I+1));
  end generate;
  SOUT <= Z(4);
```

```
end NETLIST1;

— separate handling of input and output —
architecture NETLIST2 of SHIFT is
  component DFF
    port (D, CLK: in bit; Q: out bit);
  end component;
  signal Z: bit_vector (1 to 3);
begin
  GF: for I in 0 to 3 generate
    GI1:  if (I = 0) generate
      U0:  DFF port map (SIN, CLK, Z(I+1));
    end generate;

    GI2:  if ((I > 0) and (I < 3)) generate
      UI: DFF port map (Z(I), CLK, Z(I+1));
    end generate;

    GI3:  if (I = 3) generate
      U3:  DFF port map (Z(I), CLK, SOUT);
    end generate;
  end generate;
end NETLIST2;
```

9.2 Use of Packages

In the previous examples component declaration always took place within an `architecture`. It could easily become tedious to repeat declarations of frequently used components in various architectures of a design. This unnecessary work can be avoided by the use of packages. Semiconductor vendors often provide packages containing complete cell libraries which could be conveniently accessed from different design entities within a design.

The desired package could be referenced with the `use` statement. A `package` itself may contain only component declarations; declarations of entities and belonging architectures may be placed in other design units.

Example:

```
architecture S of COMPARE is
  component XR2
    port (X, Y: in bit; Z: out bit);
  end component;

  component INV
    port (X: in bit; Z: out bit);
  end component;
  signal I: bit;
begin
  U0:  XR2 port map (A, B, I);
  U1:  INV port map (I, C);
end S;

— is equivalent to —
```

```
package XYZ_GATES                package with declarations of components
  component XR2
    port (X, Y: in bit; Z: out bit);
  end component;
  component INV
    port (X: in bit; Z: out bit);
  end component;
end XYZ_GATES;

use WORK.XYZ_GATES.ALL;                    use the XYZ_GATES package
architecture T of COMPARE is
  signal I: bit;
begin
  U0:  XR2 port map (A, B, I);
  U1:  INV port map (I, C);
end T;
```

9.3 Configurations

One of the useful features in VHDL is that it allows different `architecture` realizations of an `entity`. This enables the design process to be more efficient through the following steps:

- Step-by-step top-down refinement (from the black-box behavior down to the structure);
- Investigation of alternatives;
- Support of versions.

In structural descriptions, configurations assign specific architectures to the components. A configuration specification may be employed in two places:

1. within an `architecture`: in form of a configuration assignment.

Syntax: `for` label: entity_name
`use entity` [lib_name.]entity_name(architecture_name);

If an explicit configuration is not specified, the last (with respect to time) analyzed `architecture`, known as a *null* configuration, is used.

Example:

```
entity XR2 is                              entity declaration
  port (X, Y: in bit; Z: out bit);
end XR2;

architecture SLOW of XR2 is              the first architecture
begin
  Z <= X xor Y after 1.0 ns;
end SLOW;

architecture FAST of XR2 is            alternative architecture
begin
  Z <= X xor Y after 0.5 ns;
end FAST;
```

```
— use of XR2 in COMPARE —
architecture U of COMPARE is
  for U0:  XR2 use entity WORK.XR2(FAST);
                                        configuration for XR2
  signal I: bit;
begin
  U0:  XR2 port map (A, B, I);          explicit configuration
  U1:  INV port map (I, C);             implicit configuration
end U;
```

2. **outside an `architecture`:** Different architectures may be selected with a `configuration` statement. In this case a configuration is treated as a separate design unit which can be analyzed and simulated. Following arrangements using the configuration statement are used:

Architecture — Entity

When an `entity` has several architectures defined for it, the desired `architecture` to be used in the simulation of a design can be selected.

Syntax:
```
configuration configuration_name of entity_name is
  for architecture_name
  end for;
end configuration_name;
```

The example shows two configurations, `CFG_ONE` and `CFG_TWO`, describing the XOR gate. For the simulation of XOR the desired architecture (`FAST` or `SLOW`) may be chosen.

Example:
```
configuration CFG_ONE of XR2 is        ONE selects FAST
  for FAST
  end for;
end CFG_ONE;

configuration CFG_TWO of XR2 is        TWO selects SLOW
  for SLOW
  end for;
end CFG_TWO;
```

Architecture — Component

In this case, the selection of an appropriate architecture is made during the description of the hierarchy. When a design is structurally described (⇒ Hierarchy) and for the instantiated components there are several corresponding architectures, a `configuration` statement may be used to specify which architecture should be associated with a particular instantiation.

Syntax:
```
configuration configuration_name of entity_name is
  for architecture_name
    for label|others|all:  comp_entity_name use
      entity [lib_name.]comp_entity_name(comp_arch_name); |
      configuration [lib_name.]configuration_name;
    end for;
```

```
    ...
  end for;
  ...
end configuration_name;
```

In the following example, `MCOMP` is a circuit where different instances of `XR2` and `INV` are used.

Example:
```
configuration CFG_TRY1 of MCOMP is
  for STRUCT
    for U0:  XR2 use entity WORK.XR2(FAST);
    end for;

    — or —
    for U0:  XR2 use configuration WORK.ONE;
    end for;

    for others:  XR2 use configuration WORK.TWO;
    end for;

    for all:  INV use configuration WORK.INV(FAST);
    end for;
  end for;
end CFG_TRY1;
```

Note:
Most of the VHDL simulators allow simulation of an entity, which instantiates components, only through the configuration. Therefore, it is necessary to include a configuration statement also in those cases, where for each component of a (hierarchical) design there exist only *one* architecture.

This is normally the case when a test environment references the actual design during the simulation. It is sufficient to include the *default* configuration statement.

Syntax:
```
configuration configuration_name of entity_name is
  for architecture_name
  end for;
end configuration_name;
```

Configurations also allow to map ports of the component declaration (`locals`) to the ports of the underlying design (`formals`). Normally, the port declaration in the component declaration is a copy of the port declaration in the submoduls' `entity`. In this case a `port map` in the configuration is not required. However, in some cases it is more efficient to map ports in the configuration section.

- For instance, *generic* cell libraries may be used during the design. Through the configuration statements the design could be then mapped onto cell libraries supplied by different vendors.

- Furthermore, it is possible to replace the elements of a design by their functional equivalents. In the example below, properly wired NAND gates

are substituted for all inverters.

Example:
```
configuration CFG_NANDY of COMPARE is
  for S
    for all:  INV use entity WORK.NAND2(BEHAVE)
      port map (A => A, B => Vcc, Z => Z);
    end for;
  end for;
end CFG_NANDY;
```

9.4 Generics

While VHDL designs are structurally connected to the environment through the input and output signals, their behavior can be altered through generic values, which are treated somewhat like variables.

For example, this mechanism allows definitions of the elements' delay times to take place outside an `entity`. Thus, in order to update the component library (for instance, when switching from 1.0 μm to 0.7 μm process), a vendor needs only to specify new transition times of the components without modifying their behavioral descriptions.

Generic values appear in the `entity` declaration prior to the declaration of input and output `ports`. They can be used as constants in the corresponding `architecture`.[9] The passing and assignment of generic values can be done in the following places:

1. a default value in the `entity` declaration
2. a default value in the `component` declaration in the `architecture` or in the `package`
3. a value can be mapped in the `architecture`, in the component instantiation
4. a value can be mapped in the configuration of the `architecture`

Generics are mapped to the actual values using this notation:
declaration_name => actual_value.

Syntax:
```
— Declaration inside an entity or a component —
generic ( generic_name :  type_name [:= default_value]{;
          generic_name :  type_name [:= default_value]} );

— Instantiation —
component_label:  component_name
  generic map (value_mapping)
  port map (signal_mapping);
```

Cell libraries usually contain generic values along with the declaration of components in separate `packages`. Furthermore, generics may be specified in configurations. This way, different configurations can be investigated, in order to examine various (min-, typ-, max-delay) implementations.

[9] In the following examples, generics are used to specify delay times in structural descritions. Similarly, they can be used as constants for the description of behavior (`process`).

Example:

```
entity XR2 is
  generic (DELAY: time := 1.0 ns);default delay time as a parameter
  port (X, Y: in bit; Z: out bit);
end XR2;
architecture GENERAL of XR2 is
begin
  Z <= X xor Y after DELAY;                    use as a constant
end GENERAL;

— instantiation with a generic values —
architecture S of COMPARE is
  signal I: bit;
  component XR2
    generic (DELAY: time);
    port (X, Y: in bit; Z: out bit);
  end component;
  ...
begin
  U0:  XR2  generic map (DELAY => 1.5 ns)
            port map (A, B, I);
  ...
end S;

— a generic value in the component declaration —
architecture S of COMPARE is
  signal I: bit;
  component XR2
    generic (DELAY: time := 1.5 ns);
    port (X, Y: in bit; Z: out bit);
  end component;
  ...
begin
  U0:  XR2 port map (A, B, I);
  ...
end S;

— a generic value in the component declaration within a package ---
package XYZ_COMPONENTS is
  component XR2
    generic (DELAY: time := 1.5 ns);
    port (X, Y: in bit; Z: out bit);
  end component;
  ...
end XYZ_COMPONENTS;

— a generic value in the configuration ---
configuration CFG_LATE of COMPARE is
  for S
    for U0:  XR2 use entity WORK.XR2(GENERAL)
      generic map (DELAY => 1.5 ns);
    end for;
  end for;
end CFG_LATE;
```

10 Packages and Libraries

A `package` is an element of VHDL that contains a collection of commonly used declarations and subprograms. A `package` can also be compiled and consequently used by more than one design or entity. Following declarations may be placed in a `package`:

- types, subtypes
- constants
- components
- subprograms (procedures and functions)

Packages can be (do not necessarily have to be) further subdivided into the *declaration* and the *body*. The *declaration* part consists of public, or visible to the rest of the design, information, such as the above mentioned declarations, which essentially define the interface for the package. The *body* contains the actual implementations, such as definitions of the objects found in the *declaration*. The advantage of keeping these two parts separate is fully realized by the ease of making changes during the reiteration of design cycles. Provided the interface is correct, only the *body* would need to be modified and re-analyzed. This is particularly beneficial in the following two cases (note that subprograms *must* be subdivided into the *declaration* and the *body*):

1. deferred constants: the name and type of a deferred constant is declared in the package *declaration* section, while the actual value assignment takes place in the package *body* section.

2. subprograms — functions and procedures: the package *declaration* section lists declarations of subprograms. Their implementations are described within the package *body* section.

Syntax:
```
package package_name is
   [type_decl]
   [subtype_decl]
   [constant_decl]
   [deferred_constant_decl]
   [subprogram_header_decl]
   [component_decl]
end [package_name];

package body package_name is
   [deferred_constant_value]
   [subprogram_body]
end [package_name];
```

Packages, or rather their declarations, can be accessed from other design units with the `use` statement. To include only one element of a package, its name item_name must be specified at the end of the `use` statement. Usually several elements should be accessible. In this case the whole package can be included by specifying `all` at the end of the `use` statement. If the packages are not located in the library `WORK` (default), then the appropriate library_name must be explicitly specified.

Syntax: [`library` library_name_list;] the default library is `WORK`
`use` [library_name.]package_name.item_name; |
[library_name.]package_name.`all`;

The desired elements of a package can be accessed by the item_name. If the name is not unique, e.g. because elements with the same name are defined in different packages, the elements must be explicitly called using the notation: [library_name.]package_name.item_name.

Example:

```
— use of constant declarations —
package MY_DEFS is
  constant UNIT_DELAY: time := 1 ns;
end MY_DEFS;

entity COMPARE is
  port (  A, B:in bit;
          C: out bit);
end COMPARE;
...
library DEMO_LIB;                    other than the default library WORK
use DEMO_LIB.MY_DEFS.all;
...
architecture DFLOW of COMPARE is
begin
  C <= not (A xor B) after UNIT_DELAY;
end DFLOW;

...– deferred constant —
package MY_DEFS is
  UNIT_DELAY: time;                                  declaration only
end MY_DEFS;

package body MY_DEFS is
  constant UNIT_DELAY: time := 1 ns;                 value assignment
end MY_DEFS;

— subprogram —
package TEMPCONV is
  function C2F (C: real) return real;                declaration only
  function F2C (F: real) return real;
end TEMPCONV;

package body TEMPCONV is
  function C2F (C: real) return real is              body of a function
  begin
```

```
        return (C * 9.0 / 5.0) + 32.0;
      end C2F;

      function F2C (F: real) return real is
      ...
    end TEMPCONV;
```

Supplementary libraries and packages are used in the VHDL design for the following purposes:

collection of declarations: as already shown in the previous examples, it is possible to collect all declarations that should be shared among different (sub)designs. Usually the default library `WORK` is used for this arrangement. Similarly, development teams or even whole departments may increase the design efficiency by utilizing other supplementary libraries.

extension of VHDL: VHDL-tools suppliers offer extensions to the "Standard VHDL" through the proprietary libraries and packages. Usually, these are additional types and functions for:

- multiple-valued logic (`std_logic_1164`) and corresponding operations;
- mathematical functions (root, exponential, trigonometric, etc.);
- routines like random function generators, queue models ...;
- conversion functions for different data types.

use of cell libraries: ASIC vendors provide their proprietary cell libraries in form of VHDL libraries. These libraries could then be used for the simulation and synthesis of structural descriptions.

The actual mapping of VHDL libraries on the file system of a computer is beyond the scope of the VHDL language. It is normally controlled by the computer system configuration files.

References

[1] BHASKER J.: *A VHDL Primer*,
Englewood Cliffs: Prentice Hall, 1995.

[2] LEHMAN, WUNDER, SELZ: *Schaltungsdesign mit VHDL*,
Poing: Franzis-Verlag, 1994.

[3] LIPSETT R., SCHAEFER C.F., USSERY C.: *VHDL: Hardware Description and Design*,
Norwell, MA: Kluwer Academic Publishers 1989.

[4] MÄDER A.: *VHDL-Kurzanleitung*,
Universität Hamburg: Fachbereich Informatik, Arbeitsbereich Technische Grundlagen der Informatik

[5] PERRY D. L.: *VHDL*,
New York: Mc Graw-Hill, 1993.

[6] *IEEE Standard VHDL Language Refenrence Manual, Std 1076-1987*
IEEE, NY, 1988

[7] *IEEE Standard VHDL Language Refenrence Manual, Std 1076-1993*
IEEE, NY, 1994

Appendix

A Package TEXTIO

```
package TEXTIO is

    type LINE is access STRING;
    type TEXT is file of STRING;
    type SIDE is (RIGHT, LEFT);
    subtype WIDTH is NATURAL;

    file INPUT:  TEXT open READ_MODE is "STD_INPUT";
    file OUTPUT: TEXT open WRITE_MODE is "STD_OUTPUT";

    procedure READ (L: inout LINE; VALUE: out BIT; GOOD: out BOOLEAN);
    procedure READ (L: inout LINE; VALUE: out BIT);
    procedure READ (L: inout LINE; VALUE: out BIT_VECTOR; GOOD: out BOOLEAN);
    procedure READ (L: inout LINE; VALUE: out BIT_VECTOR);
    procedure READ (L: inout LINE; VALUE: out BOOLEAN; GOOD: out BOOLEAN);
    procedure READ (L: inout LINE; VALUE: out BOOLEAN);
    procedure READ (L: inout LINE; VALUE: out CHARACTER; GOOD: out BOOLEAN);
    procedure READ (L: inout LINE; VALUE: out CHARACTER);
    procedure READ (L: inout LINE; VALUE: out INTEGER; GOOD: out BOOLEAN);
    procedure READ (L: inout LINE; VALUE: out INTEGER);
    procedure READ (L: inout LINE; VALUE: out REAL; GOOD: out BOOLEAN);
    procedure READ (L: inout LINE; VALUE: out REAL);
    procedure READ (L: inout LINE; VALUE: out STRING; GOOD: out BOOLEAN);
    procedure READ (L: inout LINE; VALUE: out STRING);
    procedure READ (L: inout LINE; VALUE: out TIME; GOOD: out BOOLEAN);
    procedure READ (L: inout LINE; VALUE: out TIME);

    procedure READLINE (F: in TEXT; L: out LINE);
    function  ENDFILE (F: in TEXT) return BOOLEAN;
    procedure WRITELINE (F: out TEXT; L: out LINE);

    procedure WRITE (L: inout LINE; VALUE: in BIT;
                     JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
    procedure WRITE (L: inout LINE; VALUE: in BIT_VECTOR;
                     JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
    procedure WRITE (L: inout LINE; VALUE: in BOOLEAN;
                     JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
    procedure WRITE (L: inout LINE; VALUE: in CHARACTER;
                     JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
    procedure WRITE (L: inout LINE; VALUE: in INTEGER;
                     JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
    procedure WRITE (L: inout LINE; VALUE: in REAL;
                     JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0;
                     DIGITS: in NATURAL := 0);
    procedure WRITE (L: inout LINE; VALUE: in STRING;
                     JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
    procedure WRITE (L: inout LINE; VALUE: in TIME;
                     JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0;
                     UNIT: in TIME := ns);

end TEXTIO;
```