1. Introduction

2. Overview

3. Constructs

4. Usage

5. Examples
History of VHDL

1981: Department of Defense, USA launches the “Very High Speed Integrated Circuits” (VHSIC) project.

1983: Request for Proposal (RFP) issued by US Air Force to develop a language for hardware design description. The winner was a team composed of Intermetrics, IBM and TI.

1985: VHDL version 7.2 made available.

1986: Initial suite of support software released. IEEE starts effort of standardizing VHDL.
History of VHDL


2002: Work on VHDL-200x started.
VHDL Tutorial: Overview

1. Introduction

2. Overview

3. Constructs

4. Usage

5. Examples
Model of a Hardware in VHDL

A hardware model in VHDL is described in two parts, the entity declaration part and the associated architecture body part.

The entity declaration gives a name to the model being developed and also declares its external interface signals (called ports): their names, modes (in, out, inout, buffer) and their types.
Model of a Hardware in VHDL

The architecture body defines the internal details of the model — including any internal signals and one or more concurrent statements that together model the hardware.

Multiple architectures are allowed to be defined for one entity — representing the design’s description at different levels of abstraction or multiple implementations of the same design specifications — although only one of them can be selected at any time for the purpose of simulation.

As mentioned earlier, there are various ways of describing the internal behaviour, and one or more coding styles from the behavioural, structural, data-flow styles may be chosen.

Please note that VHDL is case-insensitive.
Example:

```
entity AND2_GATE is
  port(I1, I2 : in BIT; O : out BIT);
end AND2_GATE;

architecture BEH_MODEL of AND2_GATE is
begin
  -- Sensitivity list is I1 and I2
  process(I1, I2)
  begin
    O <= I1 and I2 after 2 ns;
  end process;
end BEH_MODEL;
```

Delay = 2 ns
entity declaration and architecture body are design units in VHDL.

A design unit is the smallest VHDL code segment that can be independently compiled. In VHDL parlance, compilation is called analysis.

The entity declaration is a primary design unit which can be analyzed independently.

The architecture body is a secondary design unit — it can be analyzed only if its corresponding primary design unit (i.e. the entity declaration) has already been analyzed.

The other design units in VHDL are:
- **package declaration** permits packaging a set of declarations (of types, signals, functions) which can be analyzed (compiled) once in a design library and invoked by other design units as required by one or more users. (primary design unit)

- **package body** contains the bodies of definitions corresponding to the declarations in the corresponding package declaration. (secondary design unit)

- **configuration declarations** are used to supply the entity-architecture names (bindings) for component instances in structural descriptions. (primary design unit)

Thus, a **VHDL** design description file may contain one or more design units and references to design units or items in packages that have already been compiled into one or more design libraries.
A VHDL design file looks like the following:

Reference Clauses
Design Unit
Design Unit
Reference Clauses
Design Unit
Reference Clauses
Design Unit

Or, as follows:

Package Declaration
Package Body
Package Declaration
Package Body
Reference Clauses
Entity Declaration
Architecture Body
Entity Declaration
Architecture Body
Configuration Declaration
How does VHDL address the hardware modeling issues discussed earlier?

- **Support for Concurrency**: VHDL provides a rich set of concurrent statements to model the concurrency in digital hardware. The execution of a concurrent statement is triggered by event(s) on one or more of its input signals to which it is sensitive (called sensitivity channels).

  The language semantics require that simulation time should not be advanced until all the concurrent statements whose sensitivity channels have had events at current time have been processed.
Delta Delay in Simulation

To support zero-delay simulation consistent with cause-effect sequentiality.

To take care of stability of sequential (feedback) circuit elements.

Current simulation time is not advanced.
VHDL Tutorial: Overview

Start Simulation

Increment Time

Update Signals  Evaluate Models

Events

End Simulation
VHDL Tutorial : Overview

- VHDL provides the following different concurrent statements:

  - process statement for behavioural descriptions.

  - block statement for structuring complex descriptions and to capture hierarchy.

  - concurrent assertion statement for checks.

  - concurrent procedure call for data-flow description.

  - concurrent signal assignment statement for data-flow descriptions.

  - component instantiation statement for structural descriptions.

  - generate statement for structural descriptions.
Support for abstraction levels:

VHDL supports design abstractions from behavioural level to gate level. It supports a variety of data types and operations on them to suit hardware modeling at different levels of abstraction.

Support for complex behaviour / functionality:

VHDL provides a rich set of procedural (sequential) statements (typical of high-level programming languages like C) to describe the behaviour / functionality of the hardware.

Procedural / algorithmic constructs for program structuring, data structuring and program control are available for describing the behaviour of the hardware procedurally.
Constructs are available for modeling delayed change(s) of outputs in response to changes in the inputs of digital hardware.

Constructs are available for identifying the direction of flow of information at the interface points of a digital system.

For passing data values among different concurrent statements (each concurrent statement is used to model a part of the hardware), VHDL provides *signals*. Language semantics require that the current value and predicted waveform elements of signals be maintained.

Time evolution of signal values (for all the signals in a model) represents the time evolution of the state of the digital system in response to stimuli.
VHDL Tutorial: Constructs

1. Introduction

2. Overview

3. Constructs

4. Usage

5. Examples
The main VHDL construct for behavioural modeling of hardware is the `process` statement (which is one of the concurrent statements of the language). The `process` statement has an associated body of sequential statements which are executed to compute and schedule output changes every time any input signal on the sensitivity channels of the `process` statement changes.

Example:
entity MUX_4 is
  port(IN1, IN2, IN3, IN4, S1, S0 : in BIT;
       ZOUT : out BIT);
end MUX_4;

architecture BEH of MUX_4 is
begin

  process(IN1, IN2, IN3, IN4, S0, S1)
  begin

    if (S1 = '0' and S0 = '0') then
      ZOUT <= IN1;
    elsif (S1 = '0' and S0 = '1') then
      ZOUT <= IN2;
    elsif (S1 = '1' and S0 = '0') then
      ZOUT <= IN3;
    else
      ZOUT <= IN4;
    end if;
  end process;

end BEH;
Every modeling language models its subject using certain value sets, operations on those value sets and defining storage locations to hold the values.

**VHDL** is a strongly-typed language.

- In **VHDL**, distinct value sets are defined through different type declarations. The language also has predefined types.

The set of predefined types (and subtypes) are contained in the **VHDL** package **STANDARD**. They are:

- Boolean, bit, character, severity_level,
- integer, natural, positive, real, time,
delay_length, string, bit_vector,
file_open_kind, file_open_status.

A listing of the package STANDARD is given in Annexure-I.

- Different operations on values of specific types are defined with the help of function declaration and definitions.

The language also provides a set of pre-defined operators for the specific pre-defined types.
VHDL creates four distinct classes of objects for storing values it manipulates. This is done through object declarations. The four object classes are:

1. Signal.
2. Variable.
3. Constant.
4. File.
Types, Operators and Objects in VHDL

- VHDL provides unlimited number of data types for characterizing the values held by signal, variable and constant.

Data types are categorized into four major categories:

1. Scalar types.
2. Composite types.
3. Access types.
4. File types.
Types in VHDL: Scalars

Values belonging to these types are ordered along a scale. Also, they are basic values that cannot be decomposed further into simpler values. These are of four different kinds:

1. Enumeration.
2. Integer.
3. Floating point.
4. Physical.
Example of enumeration type.

```vhdl
type THREE_LEVEL_LOGIC is ('X', '0', '1');
signal A : THREE_LEVEL_LOGIC;
```

```vhdl
type COLOUR is (green, yellow, red);
variable LIGHT : COLOUR;
```

Example of integer type.

```vhdl
type WORD_LENGTH is range 31 downto 0;
subtype DATAWORD is WORD_LENGTH range 15 downto 0;
signal DATA : DATAWORD;
```

```vhdl
variable INDEX : integer;
```

Example of floating-point type.

```vhdl
type REAL_DATA is range 0.0 to 20.0;
variable WIDTH : REAL_DATA range 1.0 to 15.0;
```

```vhdl
variable TEMPERATURE : real;
```
Example of physical type.

type time is range -2147483647 to 2147483647
  units
    fs;
    ps = 1000 fs;
    ns = 1000 ps;
    us = 1000 ns;
    ms = 1000 us;
    sec = 1000 ms;
    min = 60 sec;
    hr = 60 min;
end units;
Types in VHDL: Composites

A composite type represents a collection of values. There are two different composite types:

1. Array.

2. Record.

Arrays are homogeneous composite types i.e. the elements of an array are all of the same type.

Records are heterogeneous composite types i.e. the elements of a record can be of different types.
Example of constrained one-dimensional array type. Here the number of elements in the array is explicitly specified in the type declaration by specifying the constraints on the range of the index.

```vhdl
type ADDRESS_WORD is array(31 downto 0) of BIT;
```

Example of declaring a variable (object) of the user-defined type ADDRESS_WORD.

```vhdl```
variable NEXT_ADDRESS : ADDRESS_WORD;
```vhdl```

Example of assignment to an entire array.

```vhdl```
NEXT_ADDRESS := X"FFFFFFFF";
```vhdl```

Example of assignment to one element of the array.

```vhdl```
NEXT_ADDRESS(0) := '1';
```vhdl```

Example of assignment to a slice of the array.

```vhdl```
NEXT_ADDRESS(7 downto 0) := X"AA";
```vhdl```

Example of unconstrained one-dimensional array type. Here no constraints are specified on the range of the index in type declaration. Hence, the number of elements in the array is not specified in the type declaration.
type BIT_vector is array(natural range<> ) of BIT;

Example of declaring a variable (object) of the pre-defined type BIT_vector.

variable ACCUMULATOR : BIT_vector(15 downto 0);

Please note that you must specify the constraints on the index of an unconstrained array type when declaring an object of that type. In view of this, the following variable declaration is illegal:

variable ACCUMULATOR : BIT_vector; -- illegal

Example of an array of array: declaring one-dimensional array type whose elements are themselves of one-dimensional array type.

This is not the same as a two-dimensional array.

type DATA_WORD is array(15 downto 0) of BIT;
type ROM is array(0 to 127) of DATA_WORD;

Example of declaring a variable (object) of ROM type.

variable CONTROL_ROM : ROM;
Example of assignment to an element of `CONTROL_ROM` which is itself an array of 16 elements of type `BIT`.

```vhdl
CONTROL_ROM(6) := X"AAAA";
```

Example of assignment to one single-bit of an element of `CONTROL_ROM`.

```vhdl
CONTROL_ROM(6)(9) := '1';
```

Example of assignment to a slice of an element of `CONTROL_ROM`.

```vhdl
CONTROL_ROM(0)(7 downto 0) := X"FF";
```

Example of constrained two-dimensional array type.

```vhdl
type CHAR_SCREEN is
    array(1 to 24, 1 to 80) of CHARACTER;

type ROW is range 1 to 24;
type COLUMN is range 1 to 80;
type CHAR_SCREEN is
    array(ROW'range, COLUMN'range) of CHARACTER;
```

Example of declaring a variable (object) of the user-defined type `CHARSCREEN`. 
variable CONSOLE : CHAR_SCREEN;

Example of assignment to an element of CONSOLE.

CONSOLE(7,4) := 'C';

Note that you can also define unconstrained two-dimensional array types. However, when declaring an object (say variable) of that type, you must constrain the index range in each dimension. You can define both constrained and unconstrained array types of more than two dimensions (in fact an arbitrary number of dimensions).

Example of declaring a record type.

type REGISTER_BANK is
   record
      F0, F1 : real;
      R0, R1 : integer;
   end record;

Example of declaring a variable (object) of the user-defined type REGISTER_BANK.

variable A_BANK : REGISTER_BANK;

Example of assignment to an element of A_BANK.
A_BANK.F0 := 4.50;

Example of assignment to A_BANK (the entire record object) using an aggregate.

A_BANK := (16.0,24.0,5,10);

Aggregates can also be used to assign to entire one-dimensional arrays, one-dimensional arrays whose elements are themselves one-dimensional arrays, two or more dimensional arrays.
Subtypes of Types

A subtype is a type with constraints. The constraint specifies the subset of values for the type. The type is called the base type of the subtype. An object is said to belong to a subtype if it is of the base type and if it satisfies the constraint. Subtype declarations are used to declare subtypes. An object can be declared to belong to either a type or a subtype. The set of operations belonging to a subtype is the same as that associated with its base type. Subtypes are useful for range checking and for imposing additional constraints on types.

Example:

```vhdl
subtype MY_INTEGER is integer range 1 to 64;
subtype MEM_ADDRESS is BIT_vector(31 downto 0);
type DIGIT is ('0','1','2','3','4',
              '5','6','7','8','9');
subtype MID_DIGIT is DIGIT range '3' to '6';
```
**Types, Operators and Objects in VHDL**

**Access Types**: Values belonging to an access type are pointers to a dynamically allocated object of some other type. We will not be discussing them any further.

**File Types**: Objects of file types represent files in the host environment. They provide a mechanism by which a VHDL design communicates with the host environment. The file type declarations and declaration of file objects will be covered later.
Operators: The operators in VHDL are classified into seven categories:

1. Logical Operators.
2. Relational Operators.
4. Adding Operators.
5. Sign Operators.
7. Miscellaneous Operators.
Logical Operators:

and, or, nand, nor, xor, xnor, not

Example:

variable SUM, A, B : BIT;
SUM := A xor B;
Relational Operators:

=, /=, <, <=, >, >=

Example:

type FOUR_LOGIC_LEVEL is ('U', '0', '1', 'Z');
FOUR_LOGIC_LEVEL('U') < FOUR_LOGIC_LEVEL('Z')

if (A < 10) then Z <= '1';

if (TEMP >= CHECK) then KOUT := '0';
Shift Operators:

Logical – sll, srl

Arithmetic – sla, sra

Rotate – rol, ror.

Example:

"1001010" sll 2 gives "0101000"

"1001010" rol 2 gives "0101010"
Adding Operators:

+, -, & (concatenation)

Example:

variable SUM1, SUM2, A, B : integer;
SUM1 := A + B;
SUM2 := A - B;
constant S1 : string := "abc";
constant S2 : string := "def";
constant S3 : string := S1 & S2; -- "abcdef"
Sign Operators:

+ and – are the two sign operators. They are predefined for any numeric type, and have their usual mathematical meaning.

Multiplying Operators:

*, /, mod (modulus), rem (remainder)

mod and rem operators operate on operands of integer types and the result is also of the same type.

Example:

\[ A \text{ rem } B = A - (A/B) \times B. \]
(-7) \text{ rem } 4 \text{ has value } -3.

The result of \text{ rem} operator has the sign of its first operand.

A \mod B = A - B \times N -- \text{ for some integer } N.

7 \mod (-4) \text{ has value } -1.

The result of \text{ mod} operator has the sign of the second operand.

What is the result of the following operation?

(-7) \mod 4 = ?

7 \text{ rem } (-4) = ?
Miscellaneous Operators:

\texttt{abs} (absolute), \texttt{**} (exponentiation)

\texttt{abs} is defined for any numeric type. Exponentiation operator is defined for the left operand to be of integer or floating-point type and the exponent to be of integer type only.
Objects in VHDL: Signals

For objects of this class, the language maintains a data structure (called *driver* of the signal) that records the time and value of each assignment to the *signal* (called transaction). Certain predicted transactions are dropped from the driver to be consistent with the delay model used. Thus, the driver of a signal maintains the current value as well as predicted or assigned future waveform elements for the signal.

Each concurrent statement that assigns to a *signal* produces a driver for the signal.

If a *signal* is assigned to by more than one concurrent statement, then it is a multi-writer signal and it must be declared as a resolved signal by specifying a resolution function in the declaration of the signal. The resolution function
takes all the values that individual drivers wish to assign to the signal and resolves them to produce a single value that is actually assigned to the signal.

Objects of signal class are created through `signal` declaration statement.

Example:

```vhdl
signal X : BIT;
signal Y : BIT := '1';
-- note the signal initialization!
signal X_BUS : BIT_vector(7 downto 0) := X"FF";
signal Z : WIRED_OR BIT;
```

Z is a resolved signal of type BIT, with WIRED_OR as the resolution function.
One can also define a resolved subtype of a type and then use the resolved subtype to declare a resolved signal.

Example:

```vhdl
subtype RESOLVED_BIT is WIRED_OR BIT;
signal Z : RESOLVED_BIT;
```
The resolution function WIRED_OR may be as follows:

```vhdl
function WIRED_OR(INPUTS : BIT_vector)
return BIT is
begin -- implicit declaration of J as integer type

    for J in INPUTS’range loop
        if INPUTS(J) = '1' then
            return '1';
        end if;
    end loop;

    return '0';

end WIRED_OR;
```
Objects of class signal are declared in :

- architecture-item-declaration zone: signals declared here are accessible throughout the architecture body, but not outside it.

- entity-item-declaration zone: signals declared here are accessible throughout all the architecture bodies associated with the entity.

- block-item-declaration zone of a concurrent block statement: signals declared here are accessible throughout the block (including any nested sub-blocks).

- in package declarations.
Creating Signal Waveforms

Phase 1 <= '0', '1' after 8 ns, '0' after 13 ns, '1' after 50 ns;

Signal Drivers

RESET <= curr@now 3@T+5ns 21@T+10ns 14@T+17ns

RESET <= 3 after 5 ns, 21 after 10 ns, 14 after 17 ns;
**Effect of Transport Delay on Signal Drivers**

```vhdl
signal RX_DATA : NATURAL;
...
process
begin
...
  RX_DATA <= transport 11 after 10 ns;
```

11@T+10ns 20@T+22ns

11@T+10ns 35@T+18ns
VHDL Tutorial: Constructs

\[ \ldots \\
RX_{\text{DATA}} \leftarrow \text{transport 20 after 22 ns;} \\
\ldots \\
RX_{\text{DATA}} \leftarrow \text{transport 35 after 18 ns;} \\
\ldots \\
\text{end process;}
\]

The following summarizes the rules for adding a new transaction to a driver when transport delay is used:

1. If the delay time of the new transaction is greater than those of all the transactions already present on the driver, then the new transaction is added at the end of the driver.
2. If the delay time of the new transaction is earlier than or equal to one or more transactions on the driver, then these transactions are deleted from the driver and the new transaction is added at the end of the driver.
Effect of Inertial Delay on Signal Drivers

process
begin
    TX_DATA <= 11 after 10 ns;
    TX_DATA <= 22 after 20 ns;
    TX_DATA <= 33 after 15 ns;
    wait; -- suspends indefinitely.
end process;
The summary of rules for adding a new transaction when inertial delay is used:

1. All transactions on a driver that are scheduled to occur at or after the delay of the new transaction are deleted (as in the transport case).

2. If the value of the new transaction is the same as the value of the transactions on the driver, the new transaction is added to the driver.

3. If the value of the new transaction is different from the values of one or more transactions on the driver, these transactions are deleted from the driver and the new transaction is added.

4. For a single signal assignment statement, if the first waveform element is added to the driver, all subsequent waveforms elements of that signal assignment are also added to the driver.
Objects in VHDL: Variables

Objects of this class only have a value. No timing information of their value changes is recorded or maintained. They are merely storage locations for values/results of expressions/intermediate results of computation/passing of parameters in sequential parts of the VHDL code (inside process statement) exactly in the same way as variables are used in any procedural language like C.

- Variables can be declared in the process-item-declaration zone: variables so declared are initialized only once — at the start of the simulation run.

They can be assigned to any where throughout the sequential part of the process statement in which they are defined (including in the nested function calls). They retain their values when processes get suspended i.e. in-between process activations.
• Variables can also be declared in the declaration zone of procedure bodies and function bodies. Variables so declared are initialized every time a call is made to the procedure or function i.e. they do not retain their values across calls.

Example:

```vhdl
variable STATUS : BIT_vector(15 downto 0);
variable SUM : integer range 0 to 10 := 5;
variable ACKNOWLEDGE : Boolean := TRUE;
```
Objects of class constant can be assigned values only once — at the time of their declaration.

There is a small exception though. It is possible for constants declared in a package not to be assigned value at the time of their declaration. Their values may be assigned in the packaged body by repeating the full constant declaration along with the value assignment there. Such constants are called deferred constants.

Example:

```vhdl
constant RISE_TIME : TIME := 10 ns;
constant BUS_WIDTH : integer := 8;
```
Sequential Statements in VHDL

Following is the list of sequential statements used in a process:

- variable assignment statement.
- signal assignment statement.
- wait statement.
- if statement.
Sequential Statements in VHDL

- case statement.

- loop statement.

- next statement.

- exit statement.

- null statement.
Sequential Statements in VHDL

- report statement.

- assertion statement.

- procedure call statement.

- return statement.
Sequential Statements in VHDL

Variable Assignment Statement

Syntax:

variable_identifier := expression;

Example:

P1 : process

-- initialization statement

    variable count : integer := 0;
begin

... 
count := count + 1;
...
wait; -- indefinite wait

end process; -- this process executes only once
VHDL Tutorial: Constructs

**Sequential Statements in VHDL**

**Signal Assignment Statement**

Syntax:

\[
\text{signal\_identifier} <= \text{[transport] value} \\
\text{[after time\_expression} \\
\{, \text{value after time\_expression}\}] \\
\]

Example:

```vhdl
signal A : BIT := '0';
signal B : BIT := '1';
P1 : process(A)
```

©CEERI, Pilani

IC Design Group
begin

  . . .

  B <= A after 10 ms;

  . . .

end process;
Sequential Statements in VHDL

Wait Statement

Syntax:

wait on sensitivity_list;
wait until boolean_expression;
wait for time_expression;

These may be combined in a single wait statement.

wait on sensitivity_list
    until boolean_expression
    for time_expression;

The execution of process is always suspended when you hit the wait statement.
process(A,B)
begin
  ...
end process;

process
begin
  wait on A,B;
end process;

process
begin
  A <= '1';
  B <= A;
  wait for 100 ns;
end process;

process
begin
  wait for 0 ns;
  B <= A;
  wait for 100 ns;
end process;

VS.

process
begin
  A <= '1';
  B <= A;
  wait for 100 ns;
end process;
Example:

wait on A, B, C;
wait until A = B;
wait for 10 ns;
wait on CLOCK for 20 ns; -- max. 20 ns wait
wait on CARRY until (SUM > 100) for 50 ms;

Example: Use of wait statement

process
  variable Temp1, Temp2 : BIT;
begin

  Temp1 := A and B;
wait for 5 ns;

Temp2 := C and D;
wait for 3 ns;

Z <= Temp1 or Temp2;
wait on A, B, C, D;

-- a process can have
-- many wait statements

end process;
Sequential Statements in VHDL

If Statement

Syntax:

if boolean-expression then
  sequential-statements

{elsif boolean-expression then
  sequential-statements}

[else sequential-statements]

end if;
The sequential statement(s) associated with the first boolean-expression which evaluates to TRUE of the `if` statement is executed even when many other boolean-expressions may evaluate to TRUE. Therefore, the order of the boolean-expressions is important.
Example:

And_Process : process

begin

    if (In1 = '0' or In2 = '0') then
        Zout <= '0' after delay;
    elsif (In1 = 'X' or In2 = 'X') then
        Zout <= 'X' after delay;
    else Zout <= '1' after delay;
    end if;

    wait on In1, In2;

end process;
Sequential Statements in VHDL

Case Statement

Syntax:

```vhdl
case discrete-expression is
  -- branch #1
  when choices => sequential-statements
  -- branch #2
  when choices => sequential-statements
  -- more branches
  ...
  -- last branch
```

©CEERI, Pilani IC Design Group
[when others => sequential-statements]
end case;

discrete-expression’s all possible discrete values should be considered and taken care of in the choices section.
Example:

Multiplexer_Process : process

begin

    case SELECTOR is
        when "00" => Zout <= In0 after delay;
        when "01" => Zout <= In1 after delay;
        when "10" => Zout <= In2 after delay;
        when "11" => Zout <= In3 after delay;
    end case;

    wait on SELECTOR, In0, In1, In2, In3;

end process;
Sequential Statements in VHDL

Loop Statement

Syntax:

[loop-label : ] loop
    sequential-statements
end loop [loop-label];

[loop-label : ] while condition loop
    sequential-statements
end loop [loop-label];
[loop_label : ] for loop-variable in range loop
    sequential-statements
end loop [loop-label];

Loops can be nested.

Example:

P1 : process

    variable A : integer := 0;
    variable B : integer := 0;

begin

    L1 : loop
        A := A + 1;
        B := 20;

    L2 : loop
        if B < (A * A) then
VHDL Tutorial: Constructs

```vhdl
exit L2;
end if;
B := B - A;
end loop L2;

exit L1 when A > 10;
end loop L1;

wait;
end process;

P1 : process
variable B : integer := 1;
begin

  L1 : for A in 1 to 10 loop
    B := 20;
```
L2 : while B >= (A * A) loop

    B := B - A;

end loop L2;

end loop L1;

wait;

end process;
Sequential Statements in VHDL

**Next Statement**

This is used only inside a loop to go to the next iteration of the loop.

Syntax:

```vhdl
next [loop-label] [when condition];
```
Example:

P1 : process

variable B : integer := 1;

begin

    L1 : for A in 1 to 10 loop

    B := 20;

    L2 : loop

        B := B - A;
        next L1 when B < (A * A);

    end loop L2;

end loop L1;

wait;

end process;
Sequential Statements in VHDL

Exit Statement

This is used only inside a loop to exit the loop completely.

Syntax:

exit [loop-label] [when condition];
Example:

SUM := 1;
J := 0;

L3 : loop
    J := J + 21;
    SUM := SUM * 10;
    if (SUM > 1000) then
        exit L3;
    -- "exit;" is also sufficient, as it
    -- exits the immediately-enclosing loop.
    end if;
end loop L3;

or

SUM := 1;
J := 0;

L3 : loop
    J := J + 21;
    SUM := SUM * 10;
    exit L3 when (SUM > 1000);
end loop L3;
**Sequential Statements in VHDL**

**Null Statement**

This statement does not cause any action to take place.

Example:

```vhdl
procedure ModTwo(X : inout integer) is

begin

  case X is
    when 0 | 1 => null;
    when others => X := X mod 2;
  end case;

end ModTwo;
```

©CEERI, Pilani

IC Design Group
Sequential Statements in VHDL

Report Statement

This is used to display a message.

Syntax:

```vhdl
report string-expression
[severity severity-level];
```

Severity levels are note, warning, error and failure. The default severity level is note.

Example:
if (CLR = 'Z') then
    report "Signal CLR has High Impedance Value";
end if;

if (CLK /= '0' and CLK /= '1') then
    report "CLK is neither '0' nor '1'"
    severity warning;
end if;
Assertion statements are useful in modeling constraints of an entity.

Syntax:

```
assert condition [report message]
    [severity severity-level];
```

Example:

```
assert Enable /= 'X'
    report "Unknown Value on Enable"
    severity error;
```
Syntax:

```
procedure-name [(association-list)];
```

association-list may be specified using positional association or named association.

Example:

```
type OP_CODE is
  (ADD, SUB, MUL, DIV, LT, LE, EQ);

... 

procedure ALU_Unit(
```
A, B : in integer;
OP : in OP_CODE;
Z : out integer;
ZCOMP : out Boolean) is
begin
  case OP is
    when ADD => Z := A + B;
    when SUB => Z := A - B;
    when MUL => Z := A * B;
    when DIV => Z := A / B;
    when LT  => ZCOMP := (A < B);
    when LE  => ZCOMP := (A <= B);
    when EQ  => ZCOMP := (A = B);
  end case;
end ALU_Unit;

-- using positional association.
ALU_Unit(D1, D2, ADD, SUM, COMP);

-- using named association.
ALU_Unit(Z => SUM, B => D2, A => D1,
         OP => ADD, ZCOMP => COMP);
This statement is used to terminate the execution of a subprogram.

Syntax:

```
return [expression];
```

For a procedure when this statement is executed, control returns to the point at which the procedure was called.

For a function, the `return` statement returns a value. This value substitutes for the function call in the calling expression.
Example:

```vhdl
function AND_Function(X, Y : in BIT)
  return BIT is
begin
  if X = '1' and Y = '1' then
    return '1';
  else
    return '0';
  end if;
end AND_Function;

...  
Z <= A or AND_Function(C, D);
...  
```
1. Introduction

2. Overview

3. Constructs

4. Usage

5. Examples
Data-Flow Modeling

This style of modeling views the digital system as a network of processing stages through which the input data flows and gets processed as it flows through them.

The main VHDL construct for data-flow modeling is the concurrent signal assignment statement. Besides, the concurrent signal assignment statement, the conditional signal assignment statement and the selected signal assignment statement are used for this style of modeling. The collection of these concurrent statements models the network of processing stages, with each statement serving as an abstraction of a processing stage i.e. processing is data-driven.
architecture Data_flow of NAND4 is

    signal S1, S2, S3 : BIT;

begin

    S1 <= I1 and I2;
    S2 <= I3 and I4;
    S3 <= S1 and S2;
    ZOUT <= not S3;

end Data_flow;

The order of statements in the above architecture body is unimportant as they are concurrent statements.
Example: Modeling a Multiplexer stage (using conditional signal assignment).

\[
ZOUT <= IN0 \text{ after } 5 \text{ ns} \\
\quad \text{when } S1 = '0' \text{ and } S0 = '0' \\
\quad \text{else IN1 after } 5 \text{ ns} \\
\quad \quad \text{when } S1 = '0' \text{ and } S0 = '1' \\
\quad \quad \text{else IN2 after } 5 \text{ ns} \\
\quad \quad \quad \text{when } S1 = '1' \text{ and } S0 = '0' \\
\quad \quad \quad \text{else IN3 after } 10 \text{ ns};
\]

Example: Modeling a Multiplexer stage (using selected signal assignment).

signal MUX_Control : BIT_vector(1 downto 0);

with MUX_Control select 
\[
ZOUT <= IN0 \text{ after } 5 \text{ ns when } "00", \\
IN1 \text{ after } 5 \text{ ns when } "01", \\
IN2 \text{ after } 5 \text{ ns when } "10", \\
IN3 \text{ after } 10 \text{ ns when } "11";
\]
Structural modeling is used to describe the construction of a piece of hardware in terms of its constituent components and their interconnections.

The basic VHDL constructs for structural modeling are:

- component declaration.

- component instantiation statement.

Component declaration declares the name of a component and its external interface signals — their names, modes and types.
Example: component declaration.

```vhdl
component AND2
  port(A0, A1 : in BIT; Z : out BIT);
end component;
```

Unlike an entity declaration, VHDL does not permit the definition of an architecture body for the component. Thus, the component only has an external interface but no behaviour/internal details. It is like a socket. Component instantiation statement defines a specific instance of usage of the component and connection of its interface signals to signals in the environment e.g. plugging in a socket on the PCB and soldering its pins to the tracks of the PCB.
Example: component instantiation.

-- positional association.
U1 : AND2 port map(S1, S2, S3);

or

-- named association.
U1 : AND2 port map(A1 => S2, A0 => S1,
                   Z => S3);
A 4-input NAND gate constructed from three 2-input AND gates and one INVERTER.

entity NAND4 is
  port(I1, I2, I3, I4 : in BIT;
       ZOUT : out BIT);
end NAND4;

architecture AND2_INVERT of NAND4 is
-- declarative zone for architecture items

signal S1, S2, S3 : BIT;

component AND2
  port(A, B : in BIT; C : out BIT);
end component;

component INVERT
  port(X : in BIT; Y : out BIT);
end component;

begin

-- concurrent statement zone. For a purely
-- structural architecture only component
-- instantiation statements are used.

  U1 : AND2 port map(I1, I2, S1);
  U2 : AND2 port map(I3, I4, S2);
  U3 : AND2 port map(S1, S2, S3);
  U4 : INVERT port map(S3, ZOUT);
end AND2_INVERT;

The above example is a valid VHDL structural model that can be successfully compiled but not simulated. This is because component instances in the architecture have no associated behaviour; they merely describe the structure e.g. a board with wired sockets, but no ICs plugged in.

In order to be able to simulate the model, we need to associate a desired entity and one of its architecture bodies with each component instance. In VHDL parlance it is called configuring a component instance. The configuration of an architecture body is complete when all the components instances in it have been configured.

Configuration (of component instances and architectures) in VHDL is achieved through the following:
- configuration specification.

- configuration declaration.

Configuration specifications are included within the architecture body itself in the architecture-item-declaration region.

Configuration declaration is a separate design unit that is used to hierarchically configure the selected architecture of an entity.

Example: Configuring an architecture body through the use of configuration specification.

architecture AND2_INVERT of NAND4 is

-- declarative zone for architecture items

    signal S1, S2, S3 : BIT;
component AND2  
    port(A, B : in BIT; C : out BIT);  
end component;

component INVERT  
    port(X : in BIT; Y : out BIT);  
end component;

-- configuring the instances.

for U1, U2, U3 : AND2  
    use entity TWO_INPUT_AND(Beh1);
for U4 : INVERT  
    use entity INVERTER(Data_flow);

begin

-- concurrent statement zone. For a purely
-- structural architecture only component
-- instantiation statements are used.

    U1 : AND2 port map(I1, I2, S1);
VHDL Tutorial: Usage

U2 : AND2 port map(I3, I4, S2);
U3 : AND2 port map(S1, S2, S3);
U4 : INVERT port map(S3, ZOUT);

end AND2_INVERT;

Example: configuring and entity and its architecture body through the use of configuration declaration.

configuration My_config of NAND4 is

for AND2_INVERT

for all : AND2
    use entity TWO_INPUT_AND(Beh1);
end for; -- AND2

for U4 : INVERT
    use entity INVERTER(Data_flow);
end for; -- INVERT

end for; -- AND2_INVERT

end My_config;
An attribute is a named characteristic of items belonging to the following classes:

- types, subtypes.
- procedures, functions.
- signals, variables, constants.
Attributes in VHDL

- entities, architectures, configurations, packages.
- components.
- statement labels.
A particular attribute of a particular item may have a value. In such a case that value may be referenced in the following manner:

```
item'attribute_identifier
```

VHDL has a number of pre-defined attributes. It also allows user-defined attributes.

**Pre-defined Attributes**

For all scalar subtypes the following attributes are pre-defined:

```
left, right, high, low
```

Example:
type BIT_position is range 15 downto 0;
type FRACTION is range -0.999 to 0.999;

For the type declarations as above,

BIT_position’left is 15
BIT_position’low is 0
FRACTION’right is 0.999
FRACTION’high is 0.999

signal X : BIT_vector(15 downto 0);

X’range ≡ 15 downto 0

The construct X’range can be used to specify the range in a looping construct such as:
for I in X’range loop

The event attribute is a pre-defined attribute of signal class objects.

S’event is a Boolean value which will be TRUE if $s$ has changed value at current simulation time.

The event attribute of signals is used to detect transitions of signals in VHDL code.
For example, if one declares

    signal CLOCK : BIT;

then the following expression can be used in VHDL code to detect the rising edge of CLOCK:

    CLOCK’event and CLOCK = ’1’

If the value of this expression is TRUE, then a rising edge of the CLOCK has occurred at the current simulation time.

Thus, one can model a positive-edge triggered D flip-flop functionality as follows:

    if (CLOCK’event and CLOCK = ’1’) then
Q <= D;
end if;

CLOCK’lastevent is the time value at which the last change in the CLOCK signal occurred. This can be used to check the setup (or hold) time.
A subprogram defines a sequential algorithm that performs a certain computation. There are two kinds of subprograms:

- **Function** is usually used to describe often-used computation that returns a single value. It executes in zero simulation time.

  Common uses are as a resolution function and for type conversion functions.

- **Procedure** is usually used to partition large behavioural descriptions into modular sections. It can return zero or more values. A procedure may or may not execute in zero simulation time, depending on whether it has a `wait` statement or not.
VHDL Tutorial: Usage

Packages and Libraries

Package declaration and the associated package body are respectively primary and secondary design units.

The package declaration and associated package body are a means of sharing a set of declarations and any associated defining bodies among multiple design units (without having to re-declare them).

A set of declarations and associated defining bodies are encapsulated as a package declaration – package body pair. This pair is analysed and stored in a design library. At the desired places in the VHDL code, the library and the specific package(s) therein can be made visible to VHDL code, thus obviating the need for their repetition.
package My_Pack is

  type COLOR is (red, green, yellow);
  type VLOG_LOGIC is ('X', '0', '1', 'Z');
  constant Rise_Time : TIME := 5 ns;
  constant Fall_Time : TIME := 3 ns;

end My_Pack;

The above package declaration needs no package body. It is self-contained in terms of necessary definitions/details.
Packages and Libraries

The following example shows the use of package body.

package My_FUNC is

  constant DELAY : TIME;
  function PARITY(A : BIT_vector) return BIT;

end My_FUNC;
Note that only one package body is allowed for a package.

package body My_FUNC is

   constant DELAY : TIME := 2 ns;

   function PARITY(A : BIT_vector)
   return BIT is
      variable X : BIT := '0';

   begin

      for i in A'range loop
         if (A(i) = '1') then
            X := not(X);
         end if;
      end loop;

      return X;

   end PARITY;

end My_FUNC;
Use of Packages and Libraries

Within a design unit in the VHDL design file (the design file comprises of a collection of design units), visibility of items contained in a package that has been analyzed into a design library can be achieved (or effected) by placing the library and use clauses before the design unit.

library work;
-- work is the default library. The
-- library clause above can be dropped.

use work.My_FUNC.all;

... entity declaration ...

library CMOS;
use CMOS.NOR2;

... entity declaration or
architecture body ...

©CEERI, Pilani IC Design Group
Block statement is a concurrent statement. By itself it has no execution semantics, rather it provides additional semantics for statements that are enclosed within it.

It is primarily used for the following purposes:

1. To disable signal drivers by using guards.

2. To create an item declaration zone with a limited scope.

3. To demarcate a portion of the design or a level in design hierarchy.
Block statements can be nested.

Any number of concurrent statements (including none) can appear within a block.
Syntax:

```
block-label : block [(guard-expression)] [is] [block-header] [block-declarations] begin [concurrent-statements] end block [block-label];
```

The block-header describes the interface of the block statement to its environment (to be discussed later).

The block-declarations are visible only within the block i.e. between block ... end block.
If a guard-expression appears in a block statement, an implicit declaration of a signal named GUARD of type Boolean occurs within the block. The value of the GUARD signal is always updated to reflect the value of the guard expression (the guard-expression must always be of type Boolean). Signal assignment statements appearing within the block statement can use this GUARD signal to enable or disable their drivers.

Example: A gated inverter.

```vhdl
-- guarded signal assignment to an ordinary signal.

Binv : block (ENABLE = '1')
begin
    Z <= guarded (not A);
end block Binv;
```

The concurrent signal assignment to Z with keyword guarded preceding it is called the "guarded assignment". Guarded assignments are the only concurrent statements whose semantics are affected by the enclosing block statement.
A guarded assignment is sensitive to signal \texttt{GUARD} and all the signals that appear in the expression following the keyword \texttt{guarded}.

If the value of the \texttt{GUARD} signal is \texttt{TRUE}, then the value of expression \texttt{not A} is assigned to the target signal \texttt{Z}, otherwise the value of the expression does not affect the value of the target signal \texttt{Z}, and \texttt{Z} continues to be driven with the old value of \texttt{(not A)}.

Every guarded assignment has an equivalent \texttt{process} statement. Here is an example:

\begin{verbatim}
BG : block (guard-expression)
  signal SIG : BIT;
begin
  SIG <= guarded waveform-elements;
end block BG;
\end{verbatim}

The equivalent process statement for the guarded assignment is:
BG : block (guard-expression)
-- signal GUARD : Boolean; -- implicit
  signal SIG : BIT;
begin
-- GUARD <= guard-expression; -- implicit
  process
  begin
    if GUARD then
      SIG <= waveform-elements;
    end if;
    wait on signals-in-waveform-elements, GUARD;
  end process;
end block BG;

Though the signal named GUARD is implicitly declared in the block statement with a guard expression, it can be used explicitly in the block statement.

Example : Explicit use of signal GUARD .

B2 : block (CLEAR = '0' and PRESET = '1')
begin
  Q <= '1' when not GUARD else '0';
end block B2;

Example: Explicit declaration of GUARD signal in the block and its use in guarded assignment.

B3 : block
  signal GUARD : Boolean;
begin
  GUARD <= CLEAR = '0' and PRESET = '1';
  Q <= guarded DIN;
end block B3;

Example: Level-sensitive (active-high) latch using block statement with a guard.
entity LATCH is
  port(D, CLOCK : in BIT; Q : out BIT);
end LATCH;

architecture LATCH_BLOCK of LATCH is
begin
  L1 : block (CLOCK'event and CLOCK = '1')
  begin
    Q <= guarded D;
  end block L1;
end LATCH_BLOCK;

Example: Edge-sensitive (positive-edge triggered) flip-flop using block statement with a guard.

entity DFF is
VHDL Tutorial: Usage

```vhdl
port(D, CLOCK : in BIT; Q : out BIT);
end DFF;

architecture DFF_BBLOCK of DFF is
begin
    L2 : block(CLOCK = '1' and not CLOCK'stable)
    begin
        Q <= guarded D;
    end block 12;
end DFF_BBLOCK;
```

**Guarded vs. Ordinary Signals**

VHDL permits declaration of guarded signals of two different kinds using the usual signal declarations appended with either the keyword `register` or `bus` just before the `;` that terminates the declaration.

Note that a guarded signal must be of a resolved type.
Example:

```vhdl
signal GUARD_SIG : Wired_AND BIT register;
signal GUARD_A : Wired_OR BIT bus;
signal ORDINARY_SIG : BIT;
```

A guarded signal behaves differently (from an ordinary signal) when assigned to by a guarded assignment.

When the implicitly declared signal GUARD becomes FALSE, the driver of the guarded assignment disconnects from the target guarded signal i.e. it does not participate in the resolution of the guarded signal’s value (contrast this with the case of an ordinary signal where under a similar circumstance the driver continues to drive the old value of the signal). In case of a guarded signal a disconnection specification can be used to specify the delay after which the driver of the signal will disconnect once signal GUARD becomes FALSE.
The syntax of the disconnection specification is:

\[
\text{disconnect guarded-signal-name : signal-type after time-expression;}
\]

Example: Disconnection specification.

\[
\text{disconnect GUARD_SIG : BIT after 5 ns;}
\]

If no disconnection time is specified for a guarded signal then, by default, one delta delay is presumed.

The two guarded signal kinds — \text{bus} and \text{register} — differ in the manner in which the value of the signal is determined when all the drivers of the signal are disconnected. For a guarded signal of kind \text{bus}, the resolution function is
still called in to resolve the value of the signal, whereas for a guarded signal of kind register the resolution function is not called and the value driven by the last driver before it disconnected is maintained for the signal. These properties make the two guarded signal kinds imitate the behaviour of their corresponding hardware namesakes.

Example : Guarded assignments to guarded signals.

use work.MY_PACK.all;

-- package MY_PACK contains functions,
-- Wired_AND and Wired_OR.

entity EXAMPLE is
  port(CLOCK : in BIT; D : in BIT);
end EXAMPLE;

architecture GUARDED_SIGNAL of EXAMPLE is

  signal REG_SIG : Wired_AND BIT register;
VHDL Tutorial: Usage

```vhdl
signal BUS_SIG : Wired_OR BIT bus;
disconnect REG_SIG : BIT after 10 ns;
disconnect BUS_SIG : BIT after 10 ns;

begin

BX : block
   (CLOCK = '1' and not CLOCK'stable)
begin

   REG_SIG <= guarded D after 5 ns;
   BUS_SIG <= guarded D after 5 ns;

end block BX;
end EXAMPLE;
```

Example: Use of block statement at elaboration time of a VHDL compiler.

```vhdl
architecture DUMMY of DUMMY is

component NOR_GATE
   generic(RISE_TIME, FALL_TIME : TIME);
```
VHDL Tutorial: Usage

```vhdl
port(S0, S1 : in BIT; Q : out BIT);
end component;

component AND2_GATE
    port(DOUT : out BIT; DIN : in BIT_vector);
end component;

for N1, N2 : NOR_GATE
    use entity work.NOR2(NOR2_DELAYS);
    generic map(PT_HL => FALL_TIME,
                  PT_LH => RISE_TIME);
    port map(S0, S1, Q);

for all : AND2_GATE
    use entity work.AND2(GENERIC_EX);
    generic map(10);
    port map(A => DIN, Z => DOUT);

signal WR, RD, RW, S1, S2 : BIT;
signal SA : BIT_vector(1 to 10);

begin
```
N1 : NOR_GATE
    generic map(2 ns, 3 ns)
    port map(WR, RD, RW);

A1 : AND2_GATE
    port map(S1, SA);

N2 : NOR_GATE
    generic map(4 ns, 6 ns)
    port map(S1, SA(2), S3);

end DUMMY;

The above structural architecture on elaboration will get translated into:

N1 : blocka
    -- block for the component instantiation.
    generic(RISE_TIME, FALL_TIME : TIME);
-- The generics of the block correspond to
-- the declaration of generics in the
-- component declaration.

generic map(RISE_TIME => 2 ns,
            FALL_TIME => 3 ns);

-- Mapping of generics from
-- component to instance

port(S0, S1 : in BIT; Q : out BIT);

-- ports of the block correspond to the
-- ports in the component declaration

port map(S0 => WR, S1 => RD, Q => RW);

-- Mapping of ports from
-- component to instance

begin
VHDL Tutorial : Usage

NOR2 : block

-- A block for the bound
-- entity’s architecture

generic(PT_HL, PT_LH : TIME);

-- generics of the block correspond to
-- the generics in the entity declaration
VHDL Tutorial: Usage

generic map(PT_HL => FALL_TIME,
            PT_LH => RISE_TIME);

-- Mapping of generics from
-- entity to component

port(A, B : in BIT; Z : out BIT);

-- ports of the block correspond to the
-- ports in the entity declaration

port map(A => S0, B => S1, Z => Q);

-- Mapping of ports from
-- entity to component

begin

-- statements in architecture body
-- appear here

    end block NOR2;
end block N1;
VHDL Tutorial: Usage

```vhdl
generic map (PT_HL => FALL_TIME,
              PT_LH => RISE_TIME);

-- Mapping of generics from
-- entity to component

port (A, B : in BIT; Z : out BIT);

-- ports of the block correspond to the
-- ports in the entity declaration

port map (A => S0, B => S1, Z => Q);

-- Mapping of ports from
-- entity to component

begin

-- statements in architecture body
-- appear here

   end block NOR2;

end block N1;
```
Concurrent statements can be conditionally selected or replicated during the elaboration phase using the generate statement.

There are two forms of the generate statement:

1. For-generation scheme: for replicating concurrent statements using a for-loop like scheme.

2. If-generation scheme: for conditionally elaborating concurrent statements.

The generate statement is interpreted during elaboration. It has no simulation semantics. It resembles a macro expansion and provides a compact description of regular structures like memories, registers and counters.
Syntax (for-generation scheme):

```vhdl
generate-label:
  for generate-identifier in discrete-range
generate
  [block-declarations
begin
  concurrent-statements
end generate [generate-label];
```

Example: Using for-generation scheme.
entity FULL_ADD4 is
port(A, B : in BIT_vector(3 downto 0);
   CIN : in BIT;
   SUM : out BIT_vector(3 downto 0);
   COUT : out BIT);
end FULL_ADD4;

architecture FOR_GENERATE of FULL_ADD4 is

   component FULL_ADDER
      port(PA, PB, PC : in BIT;
           PSUM, PCOUT: out BIT);
   end component;

   signal CAR : BIT_vector(4 downto 0);
begin
  CAR(0) <= CIN;

  GK : for K in 3 downto 0 generate
      FA : FULL_ADDER port map(A(K),
        B(K), CAR(K), SUM(K), CAR(K+1));
  end generate GK;

  COUT <= CAR(4);
end FOR_GENERATE;
Example: Another usage of for-generation scheme.

```
CAR(0) <= CIN;

G2 : for M in 3 downto 0 generate
    SUM(M) <= A(M) xor B(M) xor CAR(M);
    CAR(M+1) <= A(M) and B(M) and CAR(M);
end generate G2;

COUT <= CAR(4);
```

Syntax (if-generation scheme):

```
generate-label : if expression generate
    [block-declarations
    begin
    concurrent-statements
end generate [generate-label];
```

The if-generate statement allows a conditional selection of concurrent statements at elaboration time.
Example: A 4-bit counter using both for- and if-generation schemes.

entity COUNTER4 is
  port(COUNT, CLOCK : in BIT;
       Q : buffer BIT_vector(0 to 3));
end COUNTER4;

architecture IF_GENERATE of COUNTER4 is
component D_FLIP_FLOP
  port(D, CLK : in BIT; Q : out BIT);
end component;
begin
  GK : for K in 0 to 3 generate
    GK0 : if K = 0 generate
      DFF : D_FLIP_FLOP
      port map(COUNT, CLOCK, Q(K));
    end generate GK0;

    GK1_3 : if K > 0 generate
      DFF : D_FLIP_FLOP
      port map(Q(K-1), CLOCK, Q(K));
    end generate GK1_3;
  end generate GK;
end IF_GENERATE;
Component instances in a generate statement can be bound to entities using a block configuration. A block configuration is defined for each range of generate labels.

configuration GENERATE_BIND of FULL_ADD is

use work.all;
-- Example of a declaration in the
-- configuration declarative part.

for FOR_GENERATE
-- A block configuration

   for GK(1) -- A block configuration
      for FA : FULL_ADDER
         use configuration WORK.FA_HA_CON;
      end for;
   end for;
end for;
for GK(2 to 3)
  for FA : FULL_ADDER
-- No explicit binding. Use defaults
-- i.e. use entity FULL_ADDER
-- in working library.
    end for;
end for;

for GK(0)
  for FA : FULL_ADDER
    use entity
      work.FULL_ADDER(FA_DATAFLOW);
    end for;
end for;

end for;
end GENERATE_BIND;
Alias Declaration

Syntax:

alias identifier [: identifier-type]
    is item-name;

A named item can be an object i.e. a constant, signal, variable or file.

Example:

variable DATA_WORD : BIT_vector(15 downto 0);
alias DATA_BUS : BIT_vector(7 downto 0) is
    DATA_WORD(15 downto 8);
alias STATUS : BIT_vector(0 to 3) is
DATA_WORD(3 downto 0);
alias RESET : BIT is DATA_WORD(4);
alias RX READY : BIT is DATA_WORD(5);

Given these declarations, DATA_BUS can be used wherever DATA_WORD(15 downto 8) is needed; RESET can be used wherever DATA_WORD(4) is needed and so on.

Assigning value to an alias name is the same thing as assigning value to the aliased name.
File Types and File Objects

Objects of file types represent files in the host environment. They provide a mechanism by which a VHDL design communicates with the host computer environment.

Syntax (of a file type is):

type file-type-name is file of type-name;

Example:

type VECTORS is file of BIT_vector;
type NAMES is file of STRING;
A file of type VECTORS has a sequence of values of type BIT_vector; a file of type NAMES has a sequence of strings as values in it.

A file object is declared using a file declaration.

Note that these files are binary files. For text files, use TEXTIO package.
Syntax (of a file object is):

```vhdl
file file-name : file-type-name is
    mode string-expression;
```

Example:

```vhdl
file VEC_FILE : VECTORS is
    in "/home/rs/asp/div.vec";
file OUTPUT : NAMES is out "stdout";
```

VEC_FILE is declared to be a file that contains a sequence of BIT_vectors and it is an input file. It is associated with the file "/home/rs/asp/div.vec" in the host environment.

Procedures and functions that are implicitly declared for each file type are:
procedure READ(F : in file-type-name;
    VALUE : out type-name);
-- Gets the next value in VALUE from file F.

procedure WRITE(F : out file-type-name;
    VALUE : in type-name);
-- Appends a given value in VALUE to file F.

function ENDFILE(F : in file-type-name)
    return BOOLEAN;
-- Returns FALSE if a read on an input file F will be successful in getting
-- another value, otherwise it returns TRUE.

procedure READ (F : in file-type-name;
    VALUE : out type-name;
    LENGTH : out NATURAL);
-- LENGTH returns the number of elements
-- of the array that was read. A file
-- cannot be opened or closed explicitly,
-- and values within a file can only be
-- accessed sequentially.
Example: Usage of files.

entity FA_TEST is end FA_TEST;

architecture IO_EXAMPLE of FA_TEST is

    component FULL_ADD
        port(CIN, A, B : in BIT;
             COUT, SUM : out BIT);
    end component;

    subtype STRING3 is BIT_vector(0 to 2);
    subtype STRING2 is BIT_vector(0 to 1);
    type IN_TYPE is file of STRING3;
    type OUT_TYPE is file of STRING2;

    file VEC_FILE : IN_TYPE is
        in "~/home/rs/asp/fadd.vec";
    file RESULT_FILE : OUT_TYPE is
        out "~/home/rs/asp/fadd.out";

    signal S : STRING3;
    signal Q : STRING2;
begin

FA : FULL_ADD port map(S(0), S(1),
                      S(2), Q(0), Q(1));

process
  constant PROPAGATION_DELAY :
    TIME := 25 ns;
  variable IN_STR : STRING3;
  variable OUT_STR : STRING2;

begin

  while(not ENDFILE(VEC_FILE)) loop
    READ(VEC_FILE, IN_STR);
    S <= IN_STR;

    wait for PROPAGATION_DELAY;

    OUT_STR := Q;
    WRITE(RESULT_FILE, OUT_STR);
  end loop;

end process;

end VHDL Tutorial : Usage;
end loop;

assert FALSE
report "Completed";

end process;
end IO_EXAMPLE;
The pre-defined TEXTIO package (given in Annexure-II) contains procedures and functions that provide a means for reading and writing text files. Thus, text files containing vectors (perhaps generated manually or by another program) can be used to drive simulations.

Similarly, the simulation outputs being monitored can be written onto text files for manual examination or use by another tool.

Examples:

use STD.TEXTIO.all;

This clause makes all the procedures and functions for text input/output contained in package TEXTIO analysed in library STD visible in the following design unit.
file VECTORS : TEXT is in "/home/rs/vec.txt";

Open file "/home/rs/vec.txt" as file object named VECTORS of file type TEXT in input mode.

Type TEXT is a pre-declared type in package TEXTIO whose declaration is as follows:

type TEXT is file of STRING;

The process in which the file is read may look as follows:

process(CLOCK)

    variable BUF : LINE;
    -- LINE is pre-declared type in package
    -- TEXTIO whose declaration is as follows:
    -- type LINE is access STRING;
-- Thus, type LINE is an access type
-- (pointer) which points to a STRING.

    variable N1_VAR : BIT_vector(0 to 3);
    variable N2_VAR : integer;

begin

    if ((CLOCK'event and CLOCK = '1') and
        (not ENDFILE(VECTORS))) then

        READLINE(VECTORS, BUF);

    -- READ() is an overloaded procedure

        READ(BUF, N1_VAR);
        READ(BUF, N2_VAR);

    -- apply values read to input signals
    -- of entity under test

        N1 <= N1_VAR;
N2 <= N2_VAR;

elsif (ENDFILE(VECTORS)) then

    report "Finished All the Vectors";

end if;
end process;
To test the VHDL model of any digital hardware we need to apply test stimuli waveforms to its inputs, capture the response waveforms at the model’s outputs and compare the captured response waveforms with the expected output waveforms.

A VHDL model written for the above purpose essentially simulates the running of a test on a hardware test setup with the Device Under Test (DUT) plugged in the DUT socket — and is typically called a test bench.
One can use VHDL not only for modeling the device (DUT) but also for modeling the Stimulus Generator, Response Capture and Response Analysis Block i.e. the entire test bench.
Example: A typical test bench template.

```vhdl
entity TEST_BENCH is
end TEST_BENCH;

architecture ILLUSTRATIVE of TEST_BENCH is

-- declare DUT in a component declaration.
-- declare local signals that will connect to the DUT’s I/O’s.

begin

-- Include a process that will generate the
-- stimulus waveforms on signals that will
-- connect to the DUT’s inputs. Instantiate
-- the DUT such that these signals connects
-- to the DUT’s inputs. Include a process
-- that will monitor and compare the captured
-- response waveforms from the DUT’s outputs
-- by having all the local signals that connect
-- to the DUT’s outputs on its sensitivity list.

end ILLUSTRATIVE;
```
1. Introduction

2. Overview

3. Constructs

4. Usage

5. Examples
Behavioural Modeling

Modeling hardware as a single functional block with an interface to the external world (defined through an entity declaration) and its functionality and timing behaviour being described in the architecture body using one or more process statements.

Where do we need behavioural modeling?

In the top-down design approach which is used for designing complex chips/systems.

In this approach, as a first step we focus on capturing the functional and timing behaviour (specifications) of the chip correctly — without worrying about how to realize them. Thus, we build a behavioural model of the chip/system and validate it.
We also need behavioural models for cells in a cell library (standard cell library or gate array library).
## Behavioural Modeling

<table>
<thead>
<tr>
<th>Requirement</th>
<th>VHDL Construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface of the chip</td>
<td>entity declaration</td>
</tr>
<tr>
<td>Behaviour of the chip</td>
<td>architecture body</td>
</tr>
<tr>
<td>Input pins whose changes stimulate Output changes</td>
<td>sensitivity list of the process statement or use wait statement</td>
</tr>
<tr>
<td>Details of the behaviour: functional and timing</td>
<td>sequential statements included in process statement</td>
</tr>
</tbody>
</table>
Example : Behavioural modeling of AOI circuit.

entity AOI is
  port(I1, I2, I3 : in BIT; Z : out BIT);
end AOI;
architecture BEHAVIOUR1 of A0I is

begin

    process(I1, I2, I3)
        variable V1, V2, V3 : BIT;

    begin

        V1 := I1 and I2;
        V2 := V1 or I3;
        V3 := not V2;

        Z <= V3 after 3 ns;

    end process;

end BEHAVIOUR1;
In the top-down design method after initially capturing the behavioural description, such as the one in the example above (comprising of a single process statement), partitioning of the behaviour into multiple interacting processes is done.

In the next step, entities can be defined that have their behaviours described by the individual processes. These entities can then be connected (via corresponding component declarations, their instantiations and configurations) to produce a structural description.

This constitutes one step downwards in the top-down design procedure. More than one downward steps are typically required before the components become realizable/synthesisable through a bottom-up step.
entity AOI is
  port(I1, I2, I3 : in BIT; Z : out BIT);
end AOI;

architecture BEHAVIOUR2 of AOI is
  
signal S1 : BIT;

begin

  P1 : process(I1, I2)
  begin
    S1 <= I1 and I2 after 1 ns;
  end process P1;

  P2 : process(S1, I3)
  begin
    Z <= S1 nor I3 after 2 ns;
  end process P2;

end BEHAVIOUR2;
entity A0I is
  port(I1, I2, I3 : in BIT; Z : out BIT);
end A0I;

architecture STRUCT1 of A0I is

  signal S1 : BIT;

component P1
  port(I1, I2 : in BIT; Z : out BIT);
end component;

-- note that we require only one kind
-- of component here as both AND and
-- NOR gates are of 2-input and
-- 1-output variety.
-- configuration specifications.

-- assuming entity-architecture pair is
-- defined for AND2 and NOR2 gates.

for U1 : P1 use entity AND2(AND2)
    port map(I1, I2, Z);

-- (AND2) specifies the architecture
-- chosen for the entity AND2.

for U2 : P1 use entity NOR2(NOR2)
    port map(I1, I2, Z);

-- (NOR2) specifies the architecture
-- chosen for the entity NOR2.

begin

    U1 : P1 port map(I1, I2, S1);
    U2 : P1 port map(S1, I3, Z);

end STRUCT1;
**Structural Modeling**

Modeling of the chip/system as an interconnection of available cells, parts, and/or components.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>VHDL Construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description of external interface of the chip</td>
<td>entity declaration</td>
</tr>
<tr>
<td>Description of list of component/part types (with pin identifiers)</td>
<td>component declaration</td>
</tr>
<tr>
<td>Description of actual placements (instantiations) of components and wiring</td>
<td>component instantiation statements (with port map)</td>
</tr>
<tr>
<td>Description of the behaviour of instanced parts/components</td>
<td>Implied* or Explicit†</td>
</tr>
</tbody>
</table>
* Implied: An entity of the same name as the component together with its associated most recently analyzed architecture supplies the behaviour for the instanced part.

† Explicit: The configuration specification or the configuration declaration serve to bind a particular instance or a chosen set of instances to specific entity-architecture pairs that would supply the behaviour for the instance.
Example 1: Bottom-up Design using one’s own cells and implicit configuration.

In the design file, incorporate VHDL models (entity-architecture pairs) for all the cells to be used.

Write the entity declaration for the hardware model under development using the bottom-up design approach.

In the (structural) architecture body, declare components with the same names and port specifications as the entity names for each of the cells.

Build the structural architecture for the design by instantiating these components. Default bindings for component instances to entity-architecture pairs are used in this case (by default, an instance of a component binds to the most recently analyzed architecture of the visible entity of the same name and port specifications as the component, if present).
entity AND2 is
  port(I1, I2 : in BIT; Z : out BIT);
end AND2;

architecture AND2 of AND2 is
begin
  Z <= I1 and I2 after 1 ns;
end AND2;

entity NOR2 is
  port(I1, I2 : in BIT; Z : out BIT);
end NOR2;

architecture NOR2 of NOR2 is
begin
  Z <= I1 nor I2 after 1 ns;
end NOR2;

entity AOI is
  port(I1, I2, I3 : in BIT; Z : out BIT);
end AOI;
architecture STRUCTURAL_VIEW of A0I is

signal S1 : BIT;

component AND2
  port(A, B : in BIT; C : out BIT);
end component;

component NOR2
  port(X, Y : in BIT; Z : out BIT);
end component;

-- default configuration applies
-- in this case.

begin
  U1 : AND2 port map(I1, I2, S1);
  U2 : NOR2 port map(S1, I3, Z);

end STRUCTURAL_VIEW;
Example 2 : Use of pre-compiled design libraries for Bottom-up structural design description.

In this case, we assume that the entity declaration and architecture body of each of the “cells” has been pre-compiled in a design library, cell_lib and component declarations with same names and port lists as each of the entities are declared in a package named cell_list which has also been compiled in the library, cell_lib.

The design file containing entity-architecture pairs for cells and corresponding component declarations contained in a package suited for Example 1 is as follows.

```vhdl
entity AND2 is
  port(I1, I2 : in BIT; Z : out BIT);
end AND2;
```
entity NOR2 is
  port(I1, I2 : in BIT; Z : out BIT);
end NOR2;

architecture AND2 of AND2 is
begin
  Z <= I1 and I2 after 1 ns;
end AND2;

architecture NOR2 of NOR2 is
begin
  Z <= I1 nor I2 after 1 ns;
end NOR2;

package cell_list is

  component AND2
    port(A, B : in BIT; C : out BIT);
  end component;

  component NOR2
    port(X, Y : in BIT; Z : out BIT);
  end component;

©CEERI, Pilani   IC Design Group 175
end component;

end cell_list;
Now, the hardware model of AOI (using a structural architecture body) can be described by the following design file.

```
library cell_lib;
use cell_lib.cell_list.all;

entity A0I is
  port(I1, I2, I3 : in BIT; Z : out BIT);
end A0I;

architecture structural_view of A0I is

  signal S1 : BIT;

begin

  U1 : AND2 port map(I1, I2, S1);
  U2 : NOR2 port map(S1, I3, Z);

end structural_view;
```
Example 3 : Explicit configuration usage.

It is possible to declare component names that are different from the entity names for the corresponding cells.

A structural description using such component names would require explicit configuration as default configuration rules would not be able to supply the configuration.

```
entity AND2 is
  port(I1, I2 : in BIT; Z : out BIT);
end AND2;

entity NOR2 is
  port(I1, I2 : in BIT; Z : out BIT);
end NOR2;
```
architecture AND2 of AND2 is begin  
   Z <= I1 and I2 after 1 ns;  
end AND2;

architecture NOR2 of NOR2 is begin  
   Z <= I1 nor I2 after 1 ns;  
end NOR2;

designation A0I is  
    port(I1, I2, I3 : in BIT; Z : out BIT);  
end A0I;

architecture structural_view of A0I is  

   signal S1 : BIT;

   component TWO_INPUT_AND  
       port(IN1, IN2: in BIT;  
            Z_OUT : out BIT);  
   end component;
component TWO_INPUT_NOR
  port(IN1, IN2 : in BIT;
       Z_OUT : out BIT);
end component;
-- configuration specification.

-- port-to-component pin
-- using named association
for U1 : TWO_INPUT_AND
    use entity work.AND2(AND2)
    port map(I1 => IN1, I2 => IN2,
             Z  => Z_OUT);

-- using positional association
for U2 : TWO_INPUT_NOR
    use entity work.NOR2(NOR2)
    port map(IN1, IN2, Z_OUT);

begin

-- component’s pin-to-signal
-- using named association
    U1 : TWO_INPUT_AND port map(IN1 => I1,
                                 IN2 => I2, Z_OUT => S1);

-- using positional association
U2 : TWO_INPUT_NOR port map(S1, I3, Z);
end structural_view;

One can also configure a list of instances of a component using a single configuration statement.

for U1, U2, U3 : TWO_INPUT_AND
  use entity work.AND2(AND2)
  port map(IN1, IN2, Z_OUT);

One can also configure all the instances of a component using a single configuration statement

for all : TWO_INPUT_NOR
  use entity work.NOR2(NOR2)
  port map(IN1, IN2, Z_OUT);

After explicitly configuring some of the instances of a component the remaining instances can be configured through a single configuration statement.
for others : TWO_INPUT_AND
    use entity work.AND2(AND2)
    port map(IN1, IN2, Z_OUT);
Example 4: Use of configuration declaration.

Note that in Example 3, the configuration of architecture `structural_view` of entity `AOI` was done through the use of configuration specifications. We now show the use of configuration declaration for configuring the architecture body.

configuration NUMBER_1 of AOI is

    for STRUCTURAL_VIEW
        for U1 : TWO_INPUT_AND
            use entity work.AND2(AND2)
            port map(IN1, IN2, Z_OUT);
        end for; -- U1

        for U2 : TWO_INPUT_NOR
            use entity work.NOR2(NOR2)
            port map(IN1, IN2, Z_OUT);
        end for; -- U2
    end for; -- STRUCTURAL_VIEW

end NUMBER_1;
Example 5: Use of generics.

Generics are used in VHDL to pass parameter values individually to different instances of the same component. For example, the gate-delay of the same gate may be different due to differences in the capacitive loadings of its outputs when it is instanced at different places in the design.

It is assumed in this example that the following VHDL file containing the vendor-supplied (or our own) library has already been analyzed into the design library, work.

```vhdl
entity AND2 is
  generic(AND2_delay : TIME);
  port(I1, I2 : in BIT; Z : out BIT);
end AND2;

entity NOR2 is
  generic(NOR2_delay : TIME);
  port(I1, I2 : in BIT; Z : out BIT);
end NOR2;
```
end NOR2;

architecture AND2 of AND2 is
begin
  Z <= I1 and I2 after AND2_delay;
end AND2;

architecture NOR2 of NOR2 is
begin
  Z <= I1 nor I2 after NOR2_delay;
end NOR2;

-----------------------------------------

entity A0I is
  port(I1, I2, I3 : in BIT; Z : out BIT); 
end A0I;
architecture structural_generics of AOI is

    signal S1 : BIT;

    component TWO_INPUT_AND
    generic(t_delay : TIME := 2 ns);
    -- has default value of 2 ns
    port(IN1, IN2 : in BIT;
        Z_OUT : out BIT);
    end component;

    component TWO_INPUT_NOR
    generic(t_delay : TIME); -- no default
    port(IN1, IN2 : in BIT;
        Z_OUT : out BIT);
    end component;

    for U1 : TWO_INPUT_AND
    use entity work.AND2;
    generic map(AND2_delay => t_delay);
    port map(I1 => IN1, I2 => IN2,
        Z => Z_OUT);
    end for;

©CEERI, Pilani  IC Design Group
VHDL Tutorial: Examples

for U2 : TWO_INPUT_NOR
  use entity work.NOR2
  generic map(NOR2_delay => t_delay)
  port map(I1 => IN1, I2 => IN2,
           Z => Z_OUT);

begin

  U1 : TWO_INPUT_AND
  generic map(4 ns) -- optional
  port map(I1, I2, S1);

  U2 : TWO_INPUT_NOR
  generic map(4 ns) -- compulsory
  port map(S1, I3, Z);

end structural_generics;
Example 6: Use of IEEE std_logic_1164.

library IEEE;
use IEEE.std_logic_1164.all;

entity AND2 is
  generic(AND2_delay : TIME);
  port(I1, I2 : in std_logic;
       Z : out std_logic);
end AND2;

... AND2’s architecture body ...
library IEEE;
use IEEE.std_logic_1164.all;

entity AOI is
  port(I1, I2, I3 : in std_logic;
       Z : out std_logic);
end AOI;

architecture structural_ieee of AOI is

  signal S1 : std_logic;

  component TWO_INPUT_AND
    generic(t_delay : TIME);
    port(IN1, IN2 : in std_logic;
         Z_OUT : out std_logic);
  end component;

  component TWO_INPUT_NOR
    generic(t_delay : TIME);
    port(IN1, IN2 : in std_logic;
         Z_OUT : out std_logic);
  end component;
for U1 : TWO_INPUT_AND
    use entity work.AND2
    generic map(AND2_delay => t_delay)
    port map(I1 => IN1, I2 => IN2,
             Z => Z_OUT);

for U2 : TWO_INPUT_NOR
    use entity work.NOR2
    generic map(NOR2_delay => t_delay)
    port map(I1 => IN1, I2 => IN2,
             Z => Z_OUT);

begin

    U1 : TWO_INPUT_AND
        generic map(4 ns)
        port map(I1, I2, S1);

    U2 : TWO_INPUT_NOR
        generic map(4 ns)
        port map(S1, I3, Z);

end structural_ieee;
Gate-Level Modeling

Choice of data representation for modeling of functionality.

- A variety of multi-valued logic models are possible:
  - Predefined type BIT: ‘0’, ‘1’
  - Four-valued Logic: (’X’, ’0’, ’1’, ’Z’)
  - Seven-valued Logic:
State-Strength model:

- Three States: '0', 'X', '1'
- Three Strengths: 'F', 'R', 'Z'

Nine-valued Logic:

("F0", "FX", "F1", "R0", "RX", "R1", "Z0", "ZX", "Z1")

Generalized state-strength model:

Three states, Any number of strengths.
• **IEEE std_logic_1164**:  

This is the industry-standard package for simulation and synthesis. It defines the type `std_ulogic` as:

```vhdl
type std_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
```

Its resolved subtype, `std_logic`, is the preferred type for modeling.

- `'U'` is output of all uninitialized storage node.

- `'-'` is “don’t care” for logic synthesis purposes.

• **Interval Logic**
IEEE standard 1076.4 (Standard VITAL ASIC Modeling Specification) provides a standard for the usage of the VHDL language and modeling style for modeling ASIC libraries.

It permits back-annotation of post-route SDF delay data onto a structural VHDL model via the use of generics in an automated way (through the development of tools for this purpose based on the standard).
Modeling of hardware as a flow of data through stages (or a network) of operators which modify the data as it flows through them.

It is a convenient way of specifying processing on data (which implies a network of logic gates).

Logic synthesis tools can take a data-flow description and convert it into a structural description (comprising of an interconnection of available gates in a target library).
Example:

In data-flow modeling, concurrent signal assignment statements are used to model the processing of inputs through a set of processing stages (modeled using operators of the language).

entity AOI is
  port(I1, I2, I3 : in BIT; Z : out BIT);
end AOI;

architecture data_flow1 of AOI is

  signal S1, S2 : BIT;

begin

  S1 <= I1 and I2;
  S2 <= S1 or I3;
  Z <= not S2;

end data_flow1;
Since the order of concurrent statements is not important, the model below is identical to the one above.

entity A0I is
  port(I1, I2, I3 : in BIT; Z : out BIT);
end A0I;

architecture data_flow2 of A0I is

  signal S1, S2 : BIT;

begin

  Z <= not S2;
  S2 <= S1 or I3;
  S1 <= I1 and I2;

end data_flow2;
Register-Transfer Level (RTL) Modeling

Modeling of hardware as a succession of elementary processing steps sequenced by a controller, such that during each step data from specific storage registers is transferred to the inputs of the processing elements and the outputs of the processing elements are stored in registers at the end of the step.

Transitions of a clock signal (which can be single-phase clock or multi-phase clock) indicate the starting-time and ending-time for a processing step.
Example 1: RTL model of a watch (mixed control and datapath).

package CLOCK_TYPES is
    type HOURS_TYPE is range 1 to 12;
    type MINUTES_TYPE is range 0 to 59;
    type SECONDS_TYPE is range 0 to 59;
end CLOCK_TYPES;

use work.CLOCK_TYPES.all;

entity WATCH is
    port(CLOCK, RESET: in BIT;
         HOURS: out HOURS_TYPE;
         MINUTES: out MINUTES_TYPE;
         SECONDS: out SECONDS_TYPE);
end WATCH;

architecture RTL_1 of WATCH is
begin

    process(CLOCK, RESET)

        variable HOURS_REG : HOURS_TYPE;

    end process;

end RTL_1;
variable MINUTES_REG: MINUTES_TYPE;
variable SECONDS_REG: SECONDS_TYPE;
variable HOURS_CARRY, MINUTES_CARRY:
    integer range 0 to 1;

begin

if (RESET = '0') then
    HOURS_REG := 12;
    MINUTES_REG := 0;
    SECONDS_REG := 0;
elsif (CLOCK'event and CLOCK = '1') then
    MINUTES_CARRY := 0;
    HOURS_CARRY := 0;
    if (SECONDS_REG /= 59) then
        SECONDS_REG := SECONDS_REG + 1;
    else
        SECONDS_REG := 0;
        MINUTES_CARRY := 1;
    end if;

    if (MINUTES_CARRY = 1) then
        if (MINUTES_REG /= 59) then
            \[\text{remaining code}\]
        end if;
    end if;
end if;
MINUTES_REG := MINUTES_REG + 1;
else
    MINUTES_REG := 0;
    HOURS_CARRY := 1;
end if;
end if;
if (HOURS_CARRY = 1) then
    if (HOURS_REG /= 12) then
        HOURS_REG := HOURS_REG + 1;
    else
        HOURS_REG := 1;
    end if;
end if;
end if;

HOURS <= HOURS_REG;
MINUTES <= MINUTES_REG;
SECONDS <= SECONDS_REG;

end process;

end RTL_1;
Example 2: Alternate RTL model of a watch.

```vhdl
package CLOCK_TYPES is
    type HOURS_TYPE is range 1 to 12;
    type MINUTES_TYPE is range 0 to 59;
    type SECONDS_TYPE is range 0 to 59;
end CLOCK_TYPES;

use work.CLOCK_TYPES.all;

entity WATCH is
    port(CLOCK, RESET: in BIT;
         HOURS: out HOURS_TYPE;
         MINUTES: out MINUTES_TYPE;
         SECONDS: out SECONDS_TYPE);
end WATCH;

architecture RTL_2 of WATCH is
begin

    process(CLOCK, RESET)
    begin

        variable HOURS_REG : HOURS_TYPE;
```
variable MINUTES_REG : MINUTES_TYPE;
variable SECONDS_REG : SECONDS_TYPE;
variable HOURS_CARRY, MINUTES_CARRY :
    integer range 0 to 1;

begin

if (RESET = '0') then
    HOURS <= 12;
    MINUTES <= 0;
    SECONDS <= 0;
elsif (CLOCK'event and CLOCK = '1') then
    MINUTES_CARRY := 0;
    HOURS_CARRY := 0;
    if (SECONDS_REG /= 59) then
        SECONDS_REG := SECONDS_REG + 1;
    else
        SECONDS_REG := 0;
        MINUTES_CARRY := 1;
    end if;
    if (MINUTES_CARRY = 1) then
        if (MINUTES_REG /= 59) then

©CEERI, Pilani  IC Design Group  204
MINUTES_REG := MINUTES_REG + 1;
else
  MINUTES_REG := 0;
  HOURS_CARRY := 1;
end if;
end if;
if (HOURS_CARRY = 1) then
  if (HOURS_REG /= 12) then
    HOURS_REG := HOURS_REG + 1;
  else
    HOURS_REG := 1;
  end if;
end if;
HOURS <= HOURS_REG;
MINUTES <= MINUTES_REG;
SECONDS <= SECONDS_REG;
end if;

end process;

der RTL_2;
Example 3: Another RTL model of a watch (separate sequential and combinatorial parts).

```vhdl
package CLOCK_TYPES is
    type HOURS_TYPE is range 1 to 12;
    type MINUTES_TYPE is range 0 to 59;
    type SECONDS_TYPE is range 0 to 59;
end CLOCK_TYPES;

use work.CLOCK_TYPES.all;

entity WATCH is
    port(CLOCK, RESET : in BIT;
         HOURS : inout HOURS_TYPE;
         MINUTES : inout MINUTES_TYPE;
         SECONDS : inout SECONDS_TYPE);
end WATCH;

architecture RTL_3 of WATCH is

    signal HOURS_next : HOURS_TYPE;
    signal MINUTES_next : MINUTES_TYPE;
```

CEERI, Pilani
signal SECONDS_next : SECONDS_TYPE;

begin

SEQ : process(CLOCK, RESET)

begin

if (RESET = '0') then
    HOURS <= 12;
    MINUTES <= 0;
    SECONDS <= 0;
elsif (CLOCK'event and CLOCK = '1') then
    HOURS <= HOURS_next;
    MINUTES <= MINUTES_next;
    SECONDS <= SECONDS_next;
end if;

end process SEQ;

COMBI : process(HOURS, MINUTES, SECONDS)

begin
variable HOURS_CARRY, MINUTES_CARRY :
    integer range 0 to 1;

MINUTES_CARRY := 0;
HOURS_CARRY := 0;
if (SECONDS /= 59) then
    SECONDS_next <= SECONDS + 1;
else
    SECONDS_next <= 0;
    MINUTES_CARRY := 1;
endif;
if (MINUTES_CARRY = 1) then
    if (MINUTES /= 59) then
        MINUTES_next <= MINUTES + 1;
    else
        MINUTES_next <= 0;
        HOURS_CARRY := 1;
    end if;
end if;
if (HOURS_CARRY = 1) then
if (HOURS /= 12) then
    HOURS_next <= HOURS + 1;
else
    HOURS_next <= 1;
end if;
end if;
end if;
end if;
end if;
end process COMBI;
end RTL_3;
Example 4 : Yet another RTL model of a watch.

```vhdl
package CLOCK_TYPES is
    type HOURS_TYPE is range 1 to 12;
    type MINUTES_TYPE is range 0 to 59;
    type SECONDS_TYPE is range 0 to 59;
end CLOCK_TYPES;

use work.CLOCK_TYPES.all;

entity WATCH is
    port(CLOCK, RESET : in BIT;
         HOURS : out HOURS_TYPE;
         MINUTES : out MINUTES_TYPE;
         SECONDS : out SECONDS_TYPE);
end WATCH;

architecture RTL_4 of WATCH is

    signal HOURS_REG : HOURS_TYPE;
    signal MINUTES_REG : MINUTES_TYPE;
    signal SECONDS_REG : SECONDS_TYPE;
```
begin

process(CLOCK, RESET)

    variable HOURS_CARRY, MINUTES_CARRY :
        integer range 0 to 1;

begin
if (RESET = '0') then
    HOURS_REG <= 12;
    MINUTES_REG <= 0;
    SECONDS_REG <= 0;
elsif (CLOCK'event and CLOCK = '1') then
    MINUTES_CARRY := 0;
    HOURS_CARRY := 0;
    if (SECONDS_REG /= 59) then
        SECONDS_REG <= SECONDS_REG + 1;
    else
        SECONDS_REG <= 0;
        MINUTES_CARRY := 1;
    end if;
else
    if (MINUTES_CARRY = 1) then

if (MINUTES_REG /= 59) then
    MINUTES_REG <= MINUTES_REG + 1;
else
    MINUTES_REG <= 0;
    HOURS_CARRY := 1;
end if;
endif;
if (HOURS_CARRY = 1) then
    if (HOURS_REG /= 12) then
        HOURS_REG <= HOURS_REG + 1;
    else
        HOURS_REG <= 1;
    endif;
end if;
end if;
end if;
end process;

HOURS <= HOURS_REG;
MINUTES <= MINUTES_REG;
SECONDS <= SECONDS_REG;
end RTL_4;
Error Checking

Checking setup, hold and recovery time violations for a positive-edge triggered D flip-flop with asynchronous active-low clear.

Example: Making use of passive processes in entity declarations for building checks.

entity D_FF is

    generic(TSETUP, THOLD,
           TRECOVERY : TIME := 0 ns);

    port(D, CLK : in BIT := '0';
         CLR : in BIT := '1';
         Q : out BIT := '0';
         QB : out BIT := '1');

begin
    CHECKS : process(D, CLK, CLR)

begin

if (not CLK'stable and CLK = '1') then
    assert D’stable(TSETUP)
    report "Setup Time Violation"
    severity warning;
end if;

if (not D’stable and CLK = '1') then
    assert CLK’stable(THOLD)
    report "Hold Time Violation"
    severity warning;
end if;

if (not CLK’stable and CLK = '1') then
    assert (CLR = 1 and CLR’stable(TRECOVERY))
    report "Recovery Time Violation"
    severity warning;
end if;

end process CHECKS;

end D_FF;
One can freely mix `process` statements, concurrent signal assignment statements / `block` statements and component instantiations / `generate` statements to build a single architecture body.

Example:

```vhdl
entity FULL_ADDER is
  port(A, B, CIN : in BIT;
       SUM, COUT : out BIT);
end FULL_ADDER;

architecture MIXED_ONE of FULL_ADDER is

component NOR3
  port(I1, I2, I3 : in BIT; O : out BIT);
end component;

  signal C1, C2, C3, HALF_SUM : BIT;
```
begin

-- behavioural

process(A, B, CIN)
begin
    C1 <= A and B;
    C2 <= A and CIN;
    C3 <= B and CIN;
end process;

-- structural

CARRY_NOR : NOR3 port map(C1, C2, C3, COUT);

-- data-flow

HALF_SUM <= A xor B;
SUM <= HALF_SUM xor CIN;

end MIXED_ONE;
entity SPECIAL_ENCODER is

    generic(DELAY : TIME := 5 ns);

    port(I1, I2, I3, I4 : in BIT;
        MSB, LSB : out BIT);

end SPECIAL_ENCODER;

architecture BEHAVIOURAL of SPECIAL_ENCODER is
begin
    process(I1, I2, I3, I4)
    variable BIT1, BIT0 : BIT;
begin
    if ((I1 or I2 or I3 or I4) = '0') then
        assert FALSE report "ALL INPUTS '0'";
    else
        if I4 = '1' then
            BIT1 := '1'; BIT0 := '1';
        elsif I3 = '1' then
            BIT1 := '1'; BIT0 := '0';
        elsif I2 = '1' then
            BIT1 := '0'; BIT0 := '1';
        else
            BIT1 := '0'; BIT0 := '0';
        end if;
        MSB <= BIT1 after DELAY;
        LSB <= BIT0 after DELAY;
    end if;
end process;
end BEHAVIOURAL;

package TEST_BENCH_DECL is

    type TWO_CHANNEL_TRACE_ELEMENT is record
        CH1 : BIT;
        CH0 : BIT;
    end record;

end package TEST_BENCH_DECL;
VHDL Tutorial: Examples

   AT : TIME;
end record;

type TWO_CHANNEL_TRACE_FILE is
   file of TWO_CHANNEL_TRACE_ELEMENT;

component SPECIAL_ENCODER_4_2
   port(I1, I2, I3, I4 : in BIT;
       MSB, LSB : out BIT);
end component;

constant BITS_IN_VECTOR : integer := 4;

subtype TEST_VECTOR is
   BIT_vector(BITS_IN_VECTOR - 1 downto 0);

end TEST_BENCH_DECL;

entity TEST_BENCH is
end TEST_BENCH;

use work.TEST_BENCH_DECL.all;
architecture TEST_BENCH_1 of TEST_BENCH is

constant NO_OF_VECTORS : integer := 5;

type VECTOR_MEMORY is
  array(1 to NO_OF_VECTORS) of TEST_VECTOR;

constant INPUT_VECTORS : VECTOR_MEMORY :=
  ("1010","0010","1100","0110","1111");

constant VECTOR_PERIOD : TIME := 100 ns;

signal IN1, IN2, IN3, IN4 : BIT;

signal OUT1, OUT2 : BIT;

for INST1 : SPECIAL_ENCODER_4_2 use
  entity work.SPECIAL_ENCODER(BEHAVIOURAL);

begin

INST1 : SPECIAL_ENCODER_4_2
  port map(I1 => IN1, I2 => IN2, I3 => IN3,
VHDL Tutorial : Examples

I4 => IN4, MSB => OUT2, LSB => OUT1);

APPLY_VECTORS : process
    begin

        for J in 1 to NO_OF_VECTORS loop
            IN1 <= INPUT_VECTORS(J)(0);
            IN2 <= INPUT_VECTORS(J)(1);
            IN3 <= INPUT_VECTORS(J)(2);
            IN4 <= INPUT_VECTORS(J)(3);
            wait for VECTOR_PERIOD;
        end loop;

        assert FALSE report "TEST COMPLETED";

        wait;

    end process;
TWO_CHANNEL_RECORDER : process(OUT1, OUT2)

file TRACE_OF_OUT12 : TWO_CHANNEL_TRACE_FILE
   is out "NEW_TRACE";

variable SAMPLE : TWO_CHANNEL_TRACE_ELEMENT;

begin

   SAMPLE.CH1 := OUT2;
   SAMPLE.CH0 := OUT1;
   SAMPLE.AT := NOW;
   write(TRACE_OF_OUT12, SAMPLE);

end process;

end TEST_BENCH_1;
Modulo-3 Counter

C0

Count = 0

C1

Count = 1

C2

Count = 0

Count = 1

Count = 1

Count = 0

Count = 0

Count = 1
Example: Modulo-3 Counter (if statement based model).

package COUNT_LOGIC is
    type STATE is (COUNT_0, COUNT_1, COUNT_2);
end COUNT_LOGIC;

use work.COUNT_LOGIC.all;

entity COUNTER_3 is
    port(CLK : in BIT;
         ENABLE : in BIT;
         RESET_BAR : in BIT;
         COUNT_ZERO : out BIT := '1';
         COUNT_ONE : out BIT := '0';
         COUNT_TWO : out BIT := '0');
end COUNTER_3;

architecture IF_BASED of COUNTER_3 is

signal PRESENT_STATE : STATE := COUNT_0;

begin
COUNTING : process(CLK, RESET_BAR)

begin

if RESET_BAR = '0' then
    PRESENT_STATE <= COUNT_0;
    COUNT_ZERO <= '1';
    COUNT_ONE <= '0';
    COUNT_TWO <= '0';

elsif (CLK = '1' and CLK'event and ENABLE = '1') then
    if (PRESENT_STATE = COUNT_0) then
        PRESENT_STATE <= COUNT_1;
        COUNT_ONE <= '1';
        COUNT_ZERO <= '0';

    elsif (PRESENT_STATE = COUNT_1) then
        PRESENT_STATE <= COUNT_2;
        COUNT_TWO <= '1';
        COUNT_ONE <= '0';

    end if;

end if;

end process;
elsif (PRESENT_STATE = COUNT_2) then
    PRESENT_STATE <= COUNT_0;
    COUNT_ZERO <= '1';
    COUNT_TWO <= '0';

end if;

end if;

end process COUNTING;

end IF_BASED;
Example : Modulo-3 Counter (case statement based model).

package COUNT_LOGIC is
    type STATE is (COUNT_0, COUNT_1, COUNT_2);
end COUNT_LOGIC;

use work.COUNT_LOGIC.all;

entity COUNTER_3 is
    port(CLK : in BIT;
         ENABLE : in BIT;
         RESET_BAR : in BIT;
         COUNT_ZERO : out BIT := '1';
         COUNT_ONE : out BIT := '0';
         COUNT_TWO : out BIT := '0');
end COUNTER_3;

architecture CASE_BASED of COUNTER_3 is

signal PRESENT_STATE : STATE := COUNT_0;

begin
 COUNTING : process(CLK, RESET_BAR)

begin

case RESET_BAR is

when '0' =>
    PRESENT_STATE <= COUNT_0;
    COUNT_ZERO <= '1';
    COUNT_ONE <= '0';
    COUNT_TWO <= '0';

when '1' =>
    case (CLK = '1' and CLK'event
        and ENABLE = '1') is

        when FALSE => NULL;
        when TRUE =>

            case PRESENT_STATE is

                when COUNT_0 =>
                    PRESENT_STATE <= COUNT_1;
COUNT_ONE <= '1';
COUNT_ZERO <= '0';

when COUNT_1 =>
  PRESENT_STATE <= COUNT_2;
  COUNT_TWO <= '1';
  COUNT_ONE <= '0';

when COUNT_2 =>
  PRESENT_STATE <= COUNT_0;
  COUNT_ZERO <= '1';
  COUNT_TWO <= '0';

end case;

end case;

end case;

end process COUNTING;

end CASE_BASED;
Example: Modulo-3 Counter (conditional signal assignment based model).

```vhdl
package COUNT_LOGIC is
    type STATE is (COUNT_0, COUNT_1, COUNT_2);
end COUNT_LOGIC;

use work.COUNT_LOGIC.all;

entity COUNTER_3 is
end COUNTER_3;

architecture COND_SIG of COUNTER_3 is

    signal PRESENT_STATE : STATE := COUNT_0;
    signal CLK, ENABLE, RESET_BAR : BIT;
    signal COUNT_ZERO, COUNT_ONE, COUNT_TWO : BIT;

begin

    PRESENT_STATE <= COUNT_0
    when (RESET_BAR='0') or (RESET_BAR='1'
        and PRESENT_STATE = COUNT_2 and
        ENABLE='1' and CLK='1' and CLK'event)
```
else
  COUNT_1 when (RESET_BAR = '1') and
  (PRESENT_STATE=COUNT_0 and ENABLE='1'
   and CLK = '1' and CLK'event)
else
  COUNT_2 when (RESET_BAR = '1') and
  (PRESENT_STATE=COUNT_1 and ENABLE='1'
   and CLK = '1' and CLK'event)
else
  PRESENT_STATE;
  COUNT_ZERO <= '1' when
    PRESENT_STATE = COUNT_0 else '0';
  COUNT_ONE <= '1' when
    PRESENT_STATE = COUNT_1 else '0';
  COUNT_TWO <= '1' when
    PRESENT_STATE = COUNT_2 else '0';

  CLK <= not(CLK) after 25 ns;
  RESET_BAR <= '0','1' after 20 ns;
ENABLE <= '0','1' after 50 ns;

end COND_SIG;
Example : Modulo-3 Counter (selected signal assignment based model).

package COUNT_LOGIC is
  type STATE is (COUNT_0, COUNT_1, COUNT_2);
  type CONDITION is (CONDITION_0, CONDITION_1,
                      CONDITION_2, CONDITION_NOTRAN);
end COUNT_LOGIC;

use work.COUNT_LOGIC.all;

entity COUNTER_3 is
end COUNTER_3;

architecture SEL_SIG of COUNTER_3 is

signal PRESENT_STATE : STATE := COUNT_0;
signal SEL_CONDITION : CONDITION;
signal OUTPUT : BIT_vector(2 downto 0);
signal CLK, ENABLE, RESET_BAR : BIT;
signal COUNT_ZERO, COUNT_ONE, COUNT_TWO : BIT;

begin
SEL_CONDITION <= CONDITION_0

when (RESET_BAR = '0') or (RESET_BAR = '1'
and PRESENT_STATE = COUNT_2 and
ENABLE = '1' and CLK = '1' and CLK'event)

else
    CONDITION_1 when (RESET_BAR = '1') and
    (PRESENT_STATE = COUNT_0 and ENABLE = '1'
    and CLK = '1' and CLK'event)

else
    CONDITION_2 when (RESET_BAR = '1') and
    (PRESENT_STATE = COUNT_1 and ENABLE = '1'
    and CLK = '1' and CLK'event)

else
    CONDITION_NOTRAN;

with SEL_CONDITION select
    PRESENT_STATE <=
        COUNT_0 when CONDITION_0,
COUNT_1 when CONDITION_1,
COUNT_2 when CONDITION_2,
PRESENT_STATE when CONDITION_NOTRAN;

with SEL_CONDITION select
  OUTPUT <=
  "001" when CONDITION_0,
  "010" when CONDITION_1,
  "100" when CONDITION_2,
  OUTPUT when CONDITION_NOTRAN;
COUNT_ZERO <= OUTPUT(0);
COUNT_ONE <= OUTPUT(1);
COUNT_TWO <= OUTPUT(2);
CLK <= not(CLK) after 25 ns;
RESET_BAR <= '0', '1' after 50 ns;
ENABLE <= '0', '1' after 100 ns,
  '0' after 300 ns;

end SEL_SIG;
Example: Modulo-3 Counter (if and case statement based model).

package COUNT_LOGIC is
  type STATE is (COUNT_0, COUNT_1, COUNT_2);
end COUNT_LOGIC;

use work.COUNT_LOGIC.all;

entity COUNTER_3 is
  port(CLK : in BIT;
       ENABLE : in BIT;
       RESET_BAR : in BIT;
       COUNT_ZERO: out BIT:='1';
       COUNT_ONE: out BIT:='0';
       COUNT_TWO : out BIT :='0');
end COUNTER_3;

architecture CASE_IF of COUNTER_3 is

signal PRESENT_STATE : STATE := COUNT_0;

begin

COUNTING : process(CLK, RESET_BAR)

begin

if RESET_BAR = '0' then
    PRESENT_STATE <= COUNT_0;
    COUNT_ZERO <= '1';
    COUNT_ONE <= '0';
    COUNT_TWO <= '0';

elsif (CLK = '1' and CLK'event
    and ENABLE = '1') then

    case PRESENT_STATE is

        when COUNT_0 =>
            PRESENT_STATE <= COUNT_1;
            COUNT_ONE <= '1';
            COUNT_ZERO <= '0';

        when COUNT_1 =>
            PRESENT_STATE <= COUNT_2;
            COUNT_TWO <= '1';

    end case;

end if;

end process COUNTING;
COUNT_ONE <= '0';

when COUNT_2 =>
    BEGIN
        COUNT_ONE <= '0';
        COUNT_TWO <= '0';
    end case;

end if;

end process COUNTING;

end CASE_IF;
Annexure-I: Package STANDARD

This is package STANDARD as defined in the VHDL-1993 Language Reference Manual.

package standard is
    type boolean is (FALSE, TRUE);
    type bit is ('0', '1');
    type character is (
        nul, soh, stx, etx, eot, enq, ack, bel,
        bs, ht, lf, vt, ff, cr, so, si,
        dle, dc1, dc2, dc3, dc4, nak, syn, etb,
        can, em, sub, esc, fsp, gsp, rsp, usp,
        ',' ',', '!','"','#','$','%','&','(','),'','+','-','.','/','0','1','2','3','4','5','6','7','8','9',':',';','<','=','>','?','@','A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z',
    );
end package;
'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
'X', 'Y', 'Z', '[', '\', ']', ', ', ' ',
```, 'a', 'b', 'c', 'd', 'e', 'f', 'g',
'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
'x', 'y', 'z', '{', '|', '}', '~', del,

-- Plus other characters from the
-- ISO 8859-1 standard.

type severity_level is
(note, warning, error, failure);
type integer is range
-2147483648 to 2147483647;
type real is range -1.0E308 to 1.0E308;
type time is range
-2147483647 to 2147483647
units
fs;
ps = 1000 fs;
ns = 1000 ps;
us = 1000 ns;
VHDL Tutorial : Annexures

ms = 1000 us;
sec = 1000 ms;
min = 60 sec;
hr = 60 min;
end units;

subtype delay_length is time
  range 0 fs to time’high;
impure function now return delay_length;
subtype natural is integer
  range 0 to integer’high;
subtype positive is integer
  range 1 to integer’high;
type string is array (positive range <>)
  of character;
type bit_vector is array (natural range <>)
  of bit;
type file_open_kind is (
  read_mode,
  write_mode,
  append_mode);
type file_open_status is (}
  open_ok,
status_error,
name_error,
mode_error);

attribute foreign : string;
end standard;
Annexure-II : 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;

  -- changed for VHDL93 syntax:
  file input : TEXT open read_mode is
    "STD_INPUT";
  file output : TEXT open write_mode is
    "STD_OUTPUT";

  -- changed for VHDL93 (and now a built-in):
  procedure READLINE(file f: TEXT; L: out LINE);
  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);

-- changed for VHDL93 syntax
-- (and now a built-in):
procedure WRITELINE(file f: TEXT;
    L: inout 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;
ANNEXURE-III: PACKAGE \texttt{STD\_LOGIC\_1164}

\begin{verbatim}
PACKAGE std_logic_1164 IS

-- logic state system (unresolved)
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);

TYPE std_ulogic_vector IS ARRAY (NATURAL RANGE <>) OF std_ulogic;

-- resolution function
FUNCTION resolved (s : std_ulogic_vector)
   RETURN std_ulogic;
\end{verbatim}
SUBTYPE std_logic IS resolved std_ulogic;

TYPE std_logic_vector IS ARRAY
    (NATURAL RANGE <>) OF std_logic;

-- common subtypes
SUBTYPE X01 IS resolved std_ulogic
    RANGE 'X' TO '1'; -- ('X','0','1')
SUBTYPE X01Z IS resolved std_ulogic
    RANGE 'X' TO 'Z'; -- ('X','0','1','Z')
SUBTYPE UX01 IS resolved std_ulogic
    RANGE 'U' TO '1'; -- ('U','X','0','1')
SUBTYPE UX01Z IS resolved std_ulogic
    RANGE 'U' TO 'Z'; -- ('U','X','0','1','Z')

-- overloaded logical operators
FUNCTION "and" (l : std_ulogic;
    r : std_ulogic) RETURN UX01;
FUNCTION "nand" (l : std_ulogic;
    r : std_ulogic) RETURN UX01;
FUNCTION "or" (l : std_ulogic;
r : std_ulogic) RETURN UX01;
FUNCTION "nor" (l : std_ulogic;
    r : std_ulogic) RETURN UX01;
FUNCTION "xor" (l : std_ulogic;
    r : std_ulogic) RETURN UX01;
function "xnor" (l : std_ulogic;
    r : std_ulogic) RETURN ux01;
FUNCTION "not" (l : std_ulogic)
    RETURN UX01;

-- vectorized overloaded logical operators
FUNCTION "and" (l, r : std_logic_vector)
    RETURN std_logic_vector;
FUNCTION "and" (l, r : std_ulogic_vector)
    RETURN std_ulogic_vector;

FUNCTION "nand" (l, r : std_logic_vector)
    RETURN std_logic_vector;
FUNCTION "nand" (l, r : std_ulogic_vector)
    RETURN std_ulogic_vector;

FUNCTION "or" (l, r : std_logic_vector)
RETURN std_logic_vector;
FUNCTION "or" (l, r : std_ulogic_vector)
RETURN std_ulogic_vector;

FUNCTION "nor" (l, r : std_logic_vector)
RETURN std_logic_vector;
FUNCTION "nor" (l, r : std_ulogic_vector)
RETURN std_logic_vector;

FUNCTION "xor" (l, r : std_logic_vector)
RETURN std_logic_vector;
FUNCTION "xor" (l, r : std_ulogic_vector)
RETURN std_logic_vector;

function "xnor" (l, r : std_logic_vector)
RETURN std_logic_vector;
function "xnor" (l, r : std_ulogic_vector)
RETURN std_ulogic_vector;

FUNCTION "not" (l : std_logic_vector)
RETURN std_logic_vector;
FUNCTION "not" (l : std_ulogic_vector)
RETURN std_ulogic_vector;

-- conversion functions
FUNCTION To_bit (s : std_ulogic;
        xmap : BIT := '0') RETURN BIT;
FUNCTION To_bitvector (s : std_logic_vector;
        xmap : BIT := '0') RETURN BIT_vector;
FUNCTION To_bitvector (s : std_ulogic_vector;
        xmap : BIT := '0') RETURN BIT_vector;
FUNCTION To_StdULogic (b : BIT)
    RETURN std_ulogic;
FUNCTION To_StdLogicVector (b : BIT_vector)
    RETURN std_logic_vector;
FUNCTION To_StdLogicVector (s : std_ulogic_vector)
    RETURN std_logic_vector;
FUNCTION To_StdULogicVector (b : BIT_vector)
    RETURN std_ulogic_vector;
FUNCTION To_StdULogicVector (s : std_logic_vector)
    RETURN std_ulogic_vector;
FUNCTION To_X01 (s : std_logic_vector)
  RETURN std_logic_vector;
FUNCTION To_X01 (s : std_ulogic_vector)
  RETURN std_ulogic_vector;
FUNCTION To_X01 (s : std_ulogic)
  RETURN X01;
FUNCTION To_X01 (b : BIT_vector)
  RETURN std_logic_vector;
FUNCTION To_X01 (b : BIT_vector)
  RETURN std_ulogic_vector;
FUNCTION To_X01 (b : BIT)
  RETURN X01;

FUNCTION To_X01Z (s : std_logic_vector)
  RETURN std_logic_vector;
FUNCTION To_X01Z (s : std_ulogic_vector)
  RETURN std_ulogic_vector;
FUNCTION To_X01Z (s : std_ulogic)
  RETURN X01Z;
FUNCTION To_X01Z (b : BIT_vector)
RETURN std_logic_vector;
FUNCTION To_X01Z (b : BIT_vector)
  RETURN std_ulogic_vector;
FUNCTION To_X01Z (b : BIT)
  RETURN X01Z;

FUNCTION To_UX01 (s : std_logic_vector)
  RETURN std_logic_vector;
FUNCTION To_UX01 (s : std_ulogic_vector)
  RETURN std_ulogic_vector;
FUNCTION To_UX01 (s : std_ulogic)
  RETURN UX01;
FUNCTION To_UX01 (b : BIT_vector)
  RETURN std_logic_vector;
FUNCTION To_UX01 (b : BIT_vector)
  RETURN std_ulogic_vector;
FUNCTION To_UX01 (b : BIT)
  RETURN UX01;

-- edge detection
FUNCTION rising_edge (SIGNAL s : std_ulogic)
  RETURN BOOLEAN;
FUNCTION falling_edge (SIGNAL s : std_ulogic)
    RETURN BOOLEAN;

-- object contains an unknown
FUNCTION Is_X (s : std_ulogic_vector)
    RETURN BOOLEAN;
FUNCTION Is_X (s : std_logic_vector)
    RETURN BOOLEAN;
FUNCTION Is_X (s : std_ulogic)
    RETURN BOOLEAN;

END std_logic_1164;


