Introduction to VHDL

Ms. Priyanka Chopkar
Assistant Professor, ETC, GHRCEM, Nagpur
Subject: Digital System Design

BASIC STRUCTURES IN VHDL

Basic building blocks of a VHDL description can be classified into five groups:

- Entity
- Architecture
- Package
- Configuration
- Library

A digital system is usually designed as a hierarchical collection modules. Each module corresponds to a design ***entity*** in VHDL. Each design entity has two parts:

BASIC STRUCTURES IN VHDL

- Entity declaration
- Architecture bodies

An ***entity declaration*** describes a component's external interface (input and output ports etc.), whereas ***architecture bodies*** describe its internal implementations. ***Packages*** define global information that can be used by several entities. A ***configuration*** binds component instances of a structure design into entity architecture pairs. It allows a designer to experiment with different variations of a design by selecting different implementations. A VHDL design consists of several library units, each of which is compiled and saved in a design ***library***.

ENTITY DECLARATIONS

The ***entity declaration*** provides an external view of a component but does not provide information about how a component is implemented. The syntax is ;

entity *entity_name* **is**
[generic (*generic_declarations*);]
[port (*port_declarations*);]
{*entity_declarative_item*{*constants, types, signals*};}
end [*entity_name*];

[ ] : square bracket denotes optional parameters.
| : vertical bar indicates a choice among alternatives.
{ } : a choice of none, one or more items can be made.

GENERIC DECLARATIONS

The ***generic_declaration*** declares constants that can be used to control the structure or behaviour of the entity. The syntax is ;

generic (

constant_name : type [:=init_value]

{;*constant_name* : type [:=init_value]}

);

where ***constant_name*** specifies the name of a generic constant, ***type*** specifies the data type of the constant, and ***init_value*** specifies an initial value for the constant.

PORT DECLARATIONS

The ***port_declaration*** specifies the input and output ports of the entity.

port (

port_name : [mode] type [:=init_value]

{; *port_name* : [mode] type [:=init_value]}

);

where ***port_name*** specifies the name of a port, ***mode*** specifies the direction of a port signal, ***type*** specifies the data type of a port, and ***init_value*** specifies an initial value for a port.

VHDL is not case sensitive, so xyz=xYz=XYZ !!!

PORT DECLARATIONS

There are four port modes :

- **in** : can only be read. It is used for input only (*can be only on the right side of the assignment*).
- **out** : can only be assigned a value. It is used for output only (*can be only on the left side of the assignment*).
- **inout** : can be read and assigned a value. It can have more than one driver (*can be both on the right and left side of the assignment*).
- **buffer** : can be read and assigned a value. It can have only one driver (*can be both on the right and left side of the assignment*).

Inout is a bidirectional port whereas buffer is a unidirectional one. The ***entity_declarative_item*** declares some ***constants***, ***types*** or ***signals*** that can be used in the implementation of the entity.

PORT DECLARATIONS

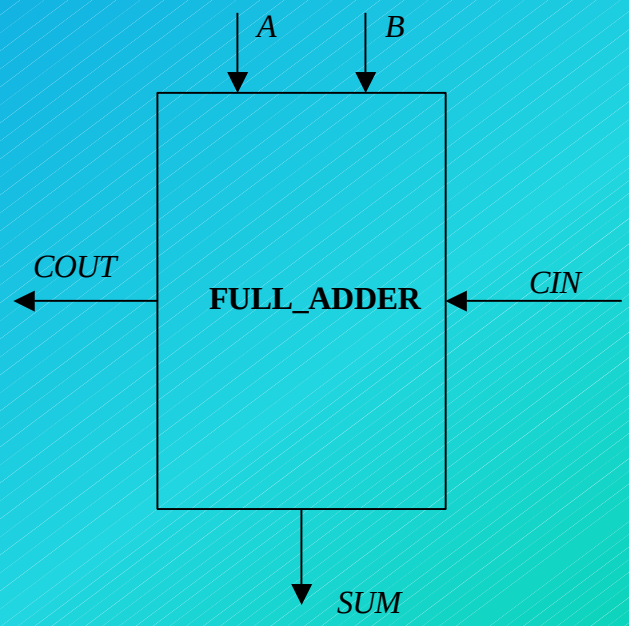
Example :

```
entity xxx is
        port (      A : in integer ;
                    B : in integer ;
                    C : out integer ;
                    D : inout integer ;
                    E : buffer integer) ;
        end xxx ;
architecture bhv of xxx is
begin
        process(A, B)
        begin
                C <= A ;        (valid : A is assigned to C)
                A <= B ;        (not valid : A is an input port so cannot be assigned a value, A is on the left side)
                E <= D + 1 ;    (valid : D is inout, so it can be both assigned and read)
                D <= C + 1 ;    (not valid : C is out port, so cannot be read for input, C is on the right side)
        end process ;
end bhv ;
```

ENTITY DECLARATION EXAMPLES

Figure-1 shows the interface of a one-bit adder. The entity name of the component is FULL_ADDER. It has input ports A, B and CIN which are of data type BIT, and output ports SUM and COUT which are also type BIT. A corresponding VHDL description is shown below.

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

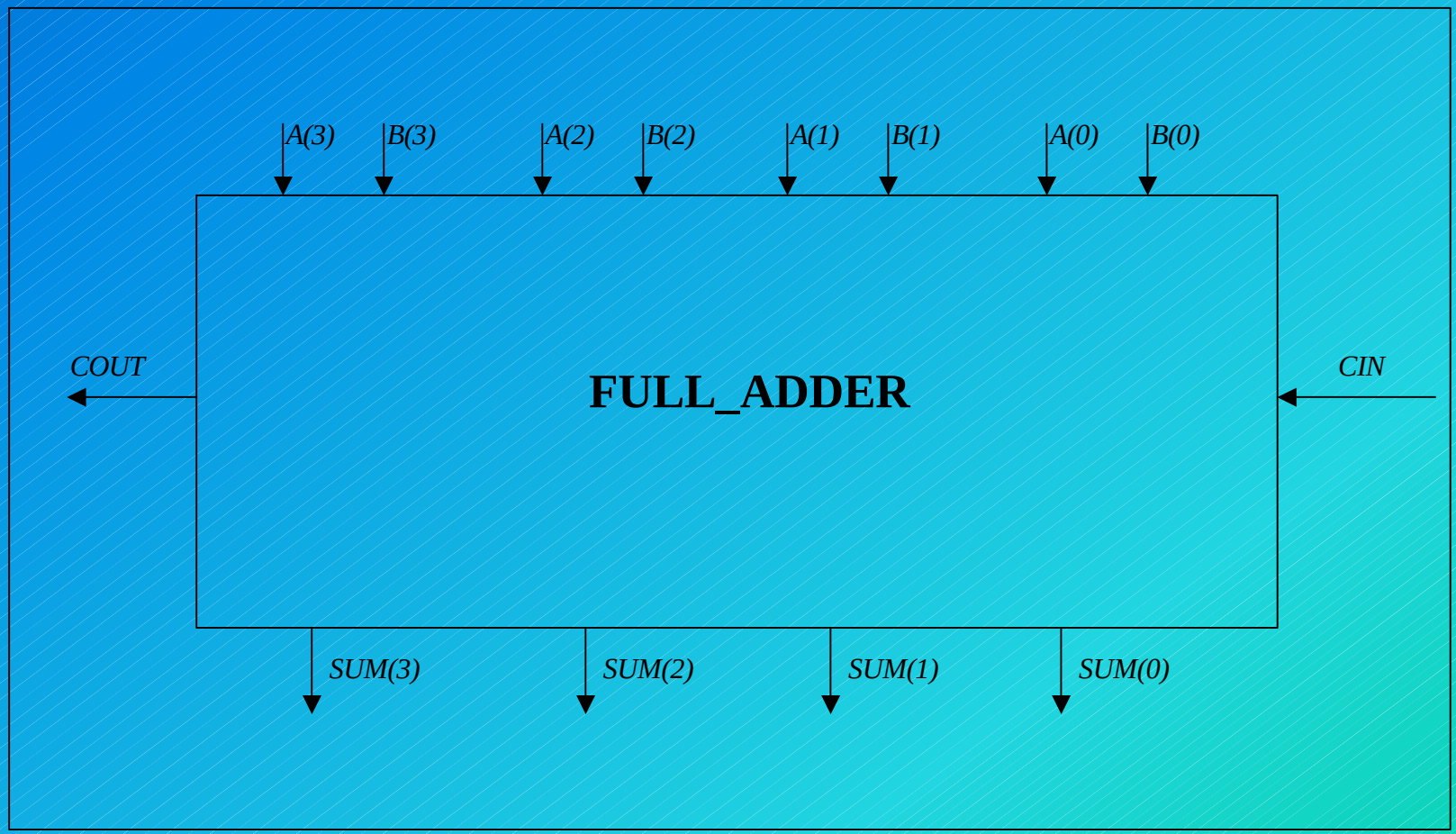


ENTITY DECLARATION EXAMPLES

We can control the structure and timing of an entity using generic constants. For example, in the following VHDL description generic constant N is used to specify the number of bits for the adder. During the simulation or the synthesis process, the actual value for each generic constant can be changed.

```
entity ADDER is
        generic ( N : INTEGER := 4 ;
                        M : TIME := 10ns );
        port (      A, B : in BIT_VECTOR (N-1 downto 0);
                    CIN : in BIT ;
                    SUM : out BIT_VECTOR (N-1 downto 0);
                    COUT : out BIT );
end ADDER ;
```

ENTITY DECLARATION EXAMPLES



ARCHITECTURES

An architecture provides an “internal” view of an entity. An entity may have more than one architecture. It defines the relationships between the inputs and the outputs of a design entity which may be expressed in terms of :

- behavioural style
- dataflow style
- structural style

An architecture determines the function of an entity. It consists of a ***declaration section*** where ***signals***, ***types***, ***constants***, ***components***, and ***subprograms*** are declared, followed by a collection of concurrent statements.

ARCHITECTURES

An architecture is declared using the following syntax :

architecture *architecture_name* **of** *entity_name* **is**
{*architecture_declarative_part*}
begin
{*concurrent_statement*}
end [*architecture_name*] ;

BEHAVIORAL STYLE ARCHITECTURES

A ***behavioural style*** specifies what a particular system does in a program like description using processes, but provides no details as to how a design is to be implemented. The primary unit of a behaviour description in VHDL is the ***process***. The example below shows a behavioural description of a full_adder.

Example :

```
architecture BEHAVIOUR of FULL_ADDER is
begin
    process (A, B, CIN)
    begin
        if   (A='0' and B='0' and CIN='0') then
             SUM <= '0';
             COUT <='0';
```

BEHAVIORAL STYLE ARCHITECTURES

```
elsif      (A='0' and B='0' and CIN='1') or
           (A='0' and B='1' and CIN='0') or
           (A='1' and B='0' and CIN='1') then
     SUM <= '1';
     COUT <='0';
elsif      (A='0' and B='1' and CIN='1') or
           (A='1' and B='0' and CIN='1') or
           (A='1' and B='1' and CIN='0') then
     SUM <= '0';
     COUT <='1';
elsif      (A='1' and B='1' and CIN='1') then
     SUM <= '1';
     COUT <='1';
   end if ;
 end process ;
end BEHAVIOUR ;
```

DATAFLOW STYLE ARCHITECTURES

A ***dataflow style*** specifies a system as a concurrent representation of the flow of control and movement of data. It models the information flow or dataflow behaviour, over time, of combinational logic functions such as adders, comparators, decoders, and primitive logic gates. The example below illustrates an architecture DATAFLOW of entity FULL_ADDER.

Example :

```
architecture DATAFLOW of FULL_ADDER is
    signal S : BIT ;
begin
    S  <= A xor B ;
    SUM  <= S xor CIN after 10ns ;
    COUT <= (A and B) or (S and CIN) after 5ns ;
end DATAFLOW ;
```

STRUCTURAL STYLE ARCHITECTURES

A ***structural style*** defines the structural implementation using component declarations and component instantiations. The following shows a structural description of the same FULL_ADDER. Two types of components are defined in this example, HALF_ADDER and OR_GATE.

Example :

```
architecture STRUCTURE of FULL_ADDER is
    component HALF_ADDER
        port ( L1, L2 : in BIT ;
               CARRY, SUM : out BIT ) ;
    end component ;
```

Structural style does not use processes !!!

STRUCTURAL STYLE ARCHITECTURES

```
component OR_GATE
        port ( L1, L2 : in BIT ;
               O : out BIT ) ;
    end component ;
    signal N1, N2, N3 : BIT ;
begin
    HA1 : HALF_ADDER port map (A, B, N1, N2) ;
    HA2 : HALF_ADDER port map (N2, CIN, N3, SUM) ;
    OR1 : OR_GATE port map (N1, N3, COUT) ;
end STRUCTURE ;
```

Top level entity consists of two HALF_ADDER instances and a OR_GATE instance. The HALF_ADDER instance can be bound to another entity which consists of an XOR gate and an AND gate.

PACKAGES

The primary purpose of a package is to collect elements that can be shared (globally) among two or more design units. It contains some common ***data types***, ***constants***, and ***subprogram*** specifications.

A package may consist of two separate design units : a ***package declaration*** and a ***package body***. A package declaration declares all the names of items that will be seen by the design units that use the package. A package body contains the implementation details of the subprograms declared in the package declaration. A package body is not required if no subprograms are declared in a package declaration.

The separation between package declaration and package body serves the same purpose as the separation between the entity declaration and architecture body.

PACKAGES

The package syntax is :

package *package_name* **is**
{*package_declarative_item*}
end [*package_name*] ;

package body *package_name* **is**
{*package_declarative_item*}
end [*package_name*] ;

Packages will be defined more in detail in latter slights !!!

PACKAGE EXAMPLE

The example below shows a package declaration. The package name is EX_PKG. Since we define a procedure called incrementer, we need to define the behaviour of the function separately in a package body.

```
package EX_PKG is
        subtype INT8 is INTEGER range 0 to 255 ;
        constant ZERO : INT8 := 0 ;
        procedure Incrementer ( variable Count : inout INT8 ) ;
end EX_PKG ;
package body EX_PKG is
        procedure Icrementer (variable Data : inout INT8) is
        begin
                if (Count >= MAX) then Count := ZERO ;
                else Count := Count + 1 ;
                end if ;
        end Incrementer ;
end EX_PKG ;
```

CONFIGURATIONS

An entity may have several architectures. During the design process, a designer may want to experiment with different variations of a design by selecting different architectures. ***Configurations*** can be used to provide fast substitutions of component instances of a structural design. The syntax is :

configuration *configuration_name* **of** *entity_name* **is**
{*configuration_declarative_part*}
for *block_specification*
{*use_clause*}
{*configuration_item*}
end for ;

CONFIGURATION EXAMPLE

For our FULL_ADDER entity we have three architectures and for structural architecture we use two HALF_ADDER's and one OR_GATE. The following example shows a configuration of entity FULL_ADDER. The name of the configuration is arbitrary (FADD_CONFIG). The STRUCTURE refers to the architecture of entity FULL_ADDER to be configured. Assume that we have already compiled HALF_ADDER and OR_GATE entities to the library *burcin* and HALF_ADDER entity has got more than one architecture one of which is STRUCTURE.

```
configuration FADD_CONFIG  of FULL_ADDER is
for STRUCTURE
      for HA1, HA2 : HALF_ADDER use entity burcin.HALF_ADDER(STRUCTURE) ;
       for OR1 : OR_GATE use entity burcin.OR_GATE ;
end for ;
end FADD_CONFIG ;
```

DESIGN LIBRARIES

The results of a VHDL compilation (*analyze*) are kept inside of a library for subsequent simulation, for use as a component in other designs. A design library can contain the following library units :

- Packages
- Entities
- Architectures
- Configurations

VHDL doesn't support hierarchical libraries. You can have as many as you want but you cannot nest them !!!

DESIGN LIBRARIES

To open a library to access a compiled entity as a part of a new VHDL design, you first need to declare the library name. The syntax is :

library *library_name* : [*path* / *directory_name*] ;

You can access compiled units from a VHDL library up to three levels of name. The syntax is :

library_name . Package_name . item_name

LIBRARY EXAMPLE

We create a package to store a constant which can be used in many designs. And we compile it to a library called *burcin*.

```
Package my_pkg is
    constant delay : time := 10ns ;
end my_pkg ;
```

Now we call *my_pkg* to use it in our design.

```
architecture DATAFLOW of FULL_ADDER is
    signal S : BIT ;
begin
    S  <= A xor B ;
    SUM  <= S xor CIN after burcin.my_pkg.delay ;
    COUT <= (A and B) or (S and CIN) after 5ns ;
end DATAFLOW ;
```

DATA OBJECTS

A ***data object*** holds a value of a specific type. There are three classes of data objects in VHDL :

- constants
- variables
- signals

The class of the object is specified by a reserved word that appears at the beginning of the declaration of that object.

CONSTANTS

A ***constant*** is an object which is initialized to a specific value when it is created and which cannot be subsequently modified. Constant declarations are allowed in ***packages***, ***entities***, ***architectures***, ***subprograms***, ***blocks***, and ***processes***. The syntax is :

constant *constant_name* {, *constant_name*} : *type* [:= *value*] ;

Examples :

```
constant YES : BOOLEAN := TRUE ;
constant CHAR7 : BIT_VECTOR (4 downto 0) := “00111” ;
constant MSB : INTEGER := 5 ;
```

VARIABLES

Variables are used to hold temporary data. They can only be declared in a ***process*** or a ***subprogram***. The syntax is :

variable *variable_name* {, *variable_name*} : *type* [:= *value*] ;

Examples :

variable X , Y : BIT ;
variable TEMP : BIT_VECTOR (8 downto 0) ;
variable DELAY : INTEGER range 0 to 15 := 5 ;

SIGNALS

Signals connect design entities together and communicates changes in values between processes. They can be interpreted as wires or busses in an actual circuit. Signals can be declared in ***packages*** *(global signals)*, ***entities*** *(entity global signals)*, ***architectures*** *(architecture global signals)* and ***blocks***. The syntax is :

signal *signal_name* {, *signal_name*} : *type* [:= *value*] ;

Examples :

```
signal BEEP : BIT  := '0' ;
signal TEMP : STD_LOGIC_VECTOR (8 downto 0) ;
signal COUNT : INTEGER range 0 to 100 := 5 ;
```

DATA TYPES

All data objects in VHDL must be defined with a ***data type***. A type declaration defines the name of the type and the range of the type. Type declarations are allowed in ***package declaration sections***, ***entity declaration sections***, ***architecture declaration sections***, ***subprogram declaration sections***, and ***process declaration sections***. Data types include :

- Enumeration types
- Integer types
- Predefined VHDL data types
- Array types
- Record types
- STD_LOGIC data type
- SIGNED and UNSIGNED data types
- Subtypes

ENUMERATION TYPES

An ***enumeration type*** is defined by listing all possible values of that type. All the values are user_defined. These values can be identifiers or single character literals. An identifier is a name such as *blue*, *ball*, *monday*. Character literals are single characters enclosed in quotes such as *'x'*, *'0'*. The syntax is :

type *type_name* **is** (*enumeration_literal* {, *enumeration_literal*}) ;

where ***type_name*** is an ***identifier*** and each ***enumeration literal*** is either an ***identifier*** or a ***character literal***.

Examples :

type COLOR is (RED, ORANGE, YELLOW, GREEN, BLUE, PURPLE) ;
type DAY is (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY) ;
type STD_LOGIC is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '_') ;

ENUMERATION TYPES

An enumeration literal can be defined in two or more enumeration types!!!

Each identifier in a type has a specific position in the type determined by the order in which the identifier appears in the type. By default RED will have a position of 0, ORANGE will have a position of 1 and so on. If we declare a data object with type COLOR and do not define an initial value, data object will be initially the default enumeration literal (*position-0*) which is RED in this case.

By default the initial value is the lowest (leftmost) value of range for that type !!!

INTEGER TYPES

Integer types are for mathematical integers. They are useful for counting, indexing, and controlling loops. In most VHDL implementations typical range is -2,147,483,647 to +2,147,483,647. The syntax is :

type *type_name* **is range** *integer_range*;

Examples :

```
type INTEGER is range -2147483647 to 2147483647 ;
type COUNT is range 0 to 10 ;
```

PREDEFINED VHDL DATA TYPES

IEEE predefined two site_specific packages : STANDART and TEXTIO in the STD library. Each contains a standard set of types and operations. The following shows a summary of data types defined in the STANDARD package.

- **BOOLEAN** : An enumeration type with two values, false and true. Logical operations and relational operations return BOOLEAN values.
- **BIT** : An enumeration type with two values, ‘0’ and ‘1’. Logical operations can take and return BIT values.
- **CHARACTER** : An enumeration type of ASCII set. Nonprinting characters are represented by a three letter name. Printable characters are represented by themselves in single quotation marks.
- **INTEGER** : Represents positive and negative numbers. Range is specified from -2,147,483,647 to +2,147,483,647 . Mathematical functions like add, subtract, multiply, divide apply to integer types.

PREDEFINED VHDL DATA TYPES

- **NATURAL** : Subtype of integers used for representing natural (non-negative) numbers.
- **POSITIVE** : Subtype of integers used for representing positive (non-negative, nonzero) numbers.
- **BIT_VECTOR** : Represents an array of BIT values.
- **STRING** : An array of CHARACTERs. A STRING value is enclosed in double quotation marks.
- **REAL** : Represents real numbers. Range is -1.0E+38 to +1.0E+38.
- **Physical type TIME** : Represents a TIME value used for simulation.

PREDEFINED VHDL DATA TYPES

Some data types defined in the STANDART package are as follows :

type BOOLEAN is (false, true);

type BIT is (‘0’, ‘1’) ;

type SEVERITY_LEVEL is (note, warning, error, failure) ;

type INTEGER is range -2147483648 to 2147483648 ;

type REAL is range -1.0E38 to 1.0E38 ;

type CHARACTER is(nul, soh, stx, 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, ‘ ‘, ‘!’, ‘””, ‘#’, ‘$’, ‘%’……..) ***(includes all keyboard characters, letters, numbers !!!)***

ARRAY TYPES

Array types group one or more elements of the same type together as a single object. There are two types of array :

- constrained array type
- unconstrained array type

A ***constrained array*** type is a type whose index range is explicitly defined. The syntax of a constrained array type is :

type *array_type_name* **is array** (*discrete_range*) **of** *subtype_indication*;

where ***array_type_name*** is the name of the constrained array type, ***discrete_range*** is a subrange of another integer type or an enumeration type, and ***subtype_indication*** is the type of each array element.

ARRAY TYPES

An ***unconstrained array*** type is a type whose index range is not defined. But index type is defined. The syntax of an unconstrained array type is :

type *array_type_name* **is array** (*type_name* **range** <>) **of** *subtype_indication*;

Example :

```
type A1 is array (0 to 31) of INTEGER ;
type BIT_VECTOR is array (NATURAL range <>) of BIT ;
type STRING is array (POSITIVE range <>) of CHARACTER ;
```

A1 is an array of 32 elements in which each element is of type *INTEGER*. The other examples show how *BIT_VECTOR* and *STRING* types are created in STANDARD package.

ARRAY TYPES

To use an unconstrained array type, the index range has to be specified!!!

Example :

```
subtype B1 is BIT_VECTOR (3 downto 0) ;
variable B2 : BIT_VECTOR (0 to 10) ;
```

Index range determines the number of elements in the array and their direction (*low* ***to*** *high* | *high* ***downto*** *low*).

ARRAY TYPES

VHDL allows declaration of multiple dimensional arrays which can be used in modelling of RAMs and ROMs.

Example :

```
type MAT is array (0 to 7, 0 to 3) of BIT  ;
constant ROM : MAT := (   ('0', '1', '0', '1') ,
                          ('1', '1', '0', '1') ,
                          ('0', '1', '1', '1') ,
                          ('0', '1', '0', '0') ,
                          ('0', '0', '0', '0') ,
                          ('1', '1', '0', '0') ,
                          ('1', '1', '1', '1') ,
                          ('1', '1', '0', '0')  ) ;
X := ROM(4,3) ;
```

X variable takes the value (0) marked with bold character.

RECORD TYPES

Record types group one or more elements of different types together as a single object. Record elements can include elements of any type, including array and records.

Example :

```
type DATE_TYPE is (SUN, MON, TUE, WED, THR, FRI, SAT) ;
type HOLIDAY is
    record
        YEAR : INTEGER range 1900 to 1999 ;
        MONTH : INTEGER range 1 to 12 ;
        DAY : INTEGER range 1 to 31 ;
        DATE : DATE_TYPE ;
    end record ;
```

RECORD TYPES

```
signal S : HOLIDAY ;
variable T1 : integer range 1900 to 1999 ;
variable T2 : DATE_TYPE ;
    T1 := S . YEAR ;
    T2 := S . DATE ;
    S . DAY <= 30 ;
```

STD_LOGIC TYPES

To model a signal line with more than two values (‘0’ and ‘1’), VHDL defines nine strengths with in a standard package. The nine values include:

```
type STD_LOGIC is ('U'--   Uninitialized
                   'X'--   Forcing unknown
                   '0' --  Forcing low
                   '1' --  Forcing high
                   'Z' --  High impedance
                   'W'--   Weak unknown
                   'L' --  Weak low
                   'H'--   Weak high
                   '_' --  Don't care
                    ) ;
```

STD_LOGIC TYPES

Similar to BIT and BIT_VECTOR types, VHDL provides STD_LOGIC_VECTOR.

To use the definitions and functions of the Standard Logic Package, the following statements have to be included in the program !!!

```
Library IEEE ;
use IEEE.STD_LOGIC_1164.all ;
```

SIGNED and UNSIGNED DATA TYPES

Both ***signed and unsinged data types*** are defined in the Standard Synthesis packages, NUMERIC_BIT and NUMERIC_STD. Objects with UNSIGNED type are interpreted as unsigned binary integers and objects with SIGNED type are interpreted as two's complement binary integers. The definitions of the data types are :

type SIGNED **is array** (NATURAL **range** <>) **of** BIT/STD_LOGIC;

type UNSIGNED **is array** (NATURAL **range** <>) **of** BIT/STD_LOGIC;

Following statements have to be included :

Library IEEE ;

use IEEE.STD_LOGIC_1164.all ;

use IEEE.NUMERIC_BIT.all ;

use IEEE.NUMERIC_BIT.all ;

SUBTYPES

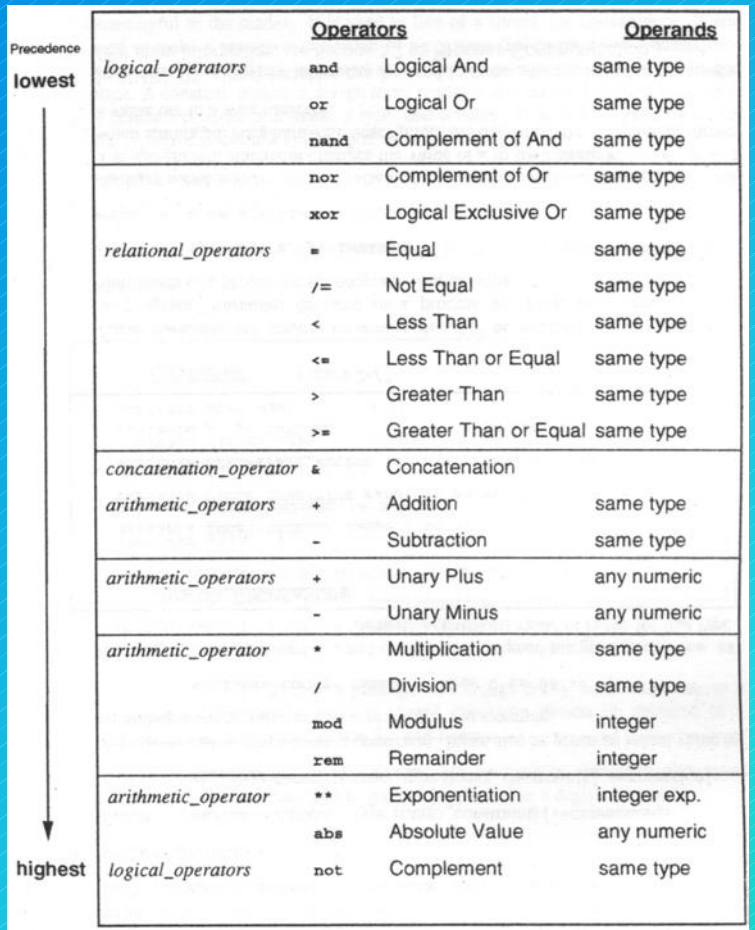
VHDL provides ***subtypes***, which are defined as subsets of other types. Anywhere a type definition can appear a subtype definition can also appear. NATURAL and POSITIVE are subtypes of INTEGER and they can be used with any INTEGER function.

Example :

```
subtype INT4 is INTEGER range 0 to 15 ;
subtype BIT_VECTOR6 is BIT_VECTOR (5 downto 0) ;
```

OPERATORS

VHDL provides six classes of ***operators***. Each operator has a precedence level. All operators in the same class have the same precedence level.



	Operators		Operands
logical_operators	`and`	Logical And	same type
	`or`	Logical Or	same type
	`nand`	Complement of And	same type
	`nor`	Complement of Or	same type
	`xor`	Logical Exclusive Or	same type
relational_operators	`=`	Equal	same type
	`/=`	Not Equal	same type
	`<`	Less Than	same type
	`<=`	Less Than or Equal	same type
	`>`	Greater Than	same type
	`>=`	Greater Than or Equal	same type
concatenation_operator	`&`	Concatenation	
arithmetic_operators	`+`	Addition	same type
	`-`	Subtraction	same type
arithmetic_operators	`+`	Unary Plus	any numeric
	`-`	Unary Minus	any numeric
arithmetic_operator	`*`	Multiplication	same type
	`/`	Division	same type
	`mod`	Modulus	integer
	`rem`	Remainder	integer
arithmetic_operator	`**`	Exponentiation	integer exp.
	`abs`	Absolute Value	any numeric
logical_operators	`not`	Complement	same type

LOGICAL OPERATORS

Logical operators and, or, nand, nor, xor, and not accept operands of pre_defined type BIT, BOOLEAN and array type of BIT. Operands must be the same type and length.

Example :

signal A, B : BIT_VECTOR (6 downto 0) ;

signal C, D, E, F, G : BIT ;

A <= B and C ; ***(not possible, operands are not the same type!!!)***

D <= (E xor F) and (C xor G) ;

RELATIONAL OPERATORS

Relational operators give a result of BOOLEAN type. Operands must be the same type and length.

Example :

```
signal A, B : BIT_VECTOR (6 downto 0) ;
signal C : BOOLEAN ;
C<= B <= A ;  (same as C<= (B <= A) ;)
```

ADDING OPERATORS

Adding operators include “+”, “-” and “&”.the concatenation operator is “&” is supported for all register array objects. It builds a register array by combining the operands. ***An unsigned (signed) number can operate with both integers and bit_vectors !!!***

Example :

```
signal W : BIT_VECTOR (3 downto 0) ;
signal X : INTEGER range 0  to 15 ;
signal Y, Z : UNSIGNED (3 downto 0) ;
    Z <= X + Y +Z ;
    Y <= Z(2 downto 0) & W(1) ;

“ABC” & “xyz” results in : “ABCxyz”
“1010” & “1” results in : “10101”
```

OPERANDS

In an expression, the operator uses the ***operands*** to compute its value. Operands can themselves be expressions. Operands in an expression include :

- Literals
- Identifiers
- Indexed names
- Slice names
- Attribute names
- Aggregates
- Qualified expressions
- Function calls
- Type conversions

LITERALS

Literals (*constants*) can be classified into two groups :

- **Scalar Type**
  - character
  - bit
  - std_logic
  - boolean
  - real
  - integer
  - time
- **Array Type**
  - string
  - bit_vector
  - std_logic_vector

CHARACTER LITERALS

A ***character literal*** defines a value by using a single character enclosed in ***single quotes*** : ‘x’. Generally VHDL is not case sensitive however it does consider case for character literals. For example , ***‘a’*** *is not the same as* ***‘A’.*** Character literal can be anything defined in the Standard package. Default value is NUL.

Example :

‘A’
‘a’
‘ ‘
‘’’

character’(‘1’)

(character literal is not the same as bit_literal ‘1’ or integer 1, so it may be necessary to provide the type name)

STRING LITERALS

A ***character string*** is an array of characters. Literal character strings are enclosed in ***double quotes***.

Example :

"A"	(array length 1)
"hold time error"	(array length 15)
" x"	
string'("10")	

BIT LITERALS

A ***bit literal*** represents two discrete values by using the character literals ‘0’ and ‘1’. Sometimes it may be necessary to make the bit type literal explicit to distinguish it from a character.

Example :

```
‘1’
‘0’
bit’(‘1’)
```

BIT_VECTOR LITERALS

A ***bit_vector literal*** is an array of bits enclosed in double quotes.

Example :

```
"00110101"
x"00FF"
b"10111"
o"277"
bit_vector'("10")
```

'x' is used for hexadecimal values, '***b'*** for binary, '***o'*** for octal.

STD_LOGIC LITERALS

A ***standard logic literal*** is one of the nine values defined in the standard package which should be given in upper case letters and single quote marks.

Example :

'U' **not** *'u'*
'X'
'0'
'1'
'Z'
'W'
'L'
'H'
'_'

STD_LOGIC_VECTOR LITERALS

A ***standard logic vector literal*** is an array of std_logic elements given in double quotes.

Example :

“10_1Z”
“UUUU”
signed’(“1011”)

BOOLEAN LITERALS

A ***boolean literal*** represents two discrete values, ***true*** or ***false***.

Example :

true

false

True

TRUE (not case sensitive)

REAL LITERALS

A ***real literal*** represents the real numbers between -1.0E+38 and 1.0E+38. Synthesis tools typically do not support either real arithmetic or real literals, but simulators do support type real.

A real number may be positive or negative, but must always be written with a decimal point !!!

Example :

+1.0 ***NOT*** *'1' or 1 or '1.0'*
0.0 ***NOT*** *0*
-1.0
-1.0E+10

INTEGER LITERALS

An ***integer literal*** represents the integer numbers between -2,147,483,647 and 2,147,483,647.

Example :

+1
862 ***NOT*** *862.0*
-257
+123_456
16#00FF#

base_n#number# means ***number*** is defined in base ***n***, where ***n*** is ***2*** to ***16***.

TIME (Physical) LITERALS

The only predefined physical type is ***time.***

Example :

10 ns
100 us
6.3 ns

It is important to separate the number from the unit of measure with at least one space !!!

IDENTIFIERS

An ***identifier*** is a simple name. It is a name for a constant, a variable, a signal, an entity, a port, a subprogram, and a parameter declaration. A name must begin with an alphabetic letter followed by letters, underscores or digits. Underscore '_' cannot be the last character. VHDL identifiers are ***not case sensitive***. There are some reserved words in VHDL such as *entity*, *port* etc. which cannot be used as an identifier.

Example :

xyx = xYZ = XYZ = XyZ
S(3) (Array element)
X3

INDEXED NAMES

An ***index name*** identifies one element of an array object. The syntax is :

array_name (*expression*)

where ***array_name*** is the name of a constant or variable of an array type. The ***expression*** must return a value within the array's index range.

Example :

```
type memory is array (0 to 7) of INTEGER range 0 to 123 ;
variable DATA_ARRAY : memory ;
variable ADDR : INTEGER range 0 to 7 ;
variable DATA : INTEGER range 0 to 123 ;
DATA := DATA_ARRAY (ADDR) ;
```

SLICE NAMES and ALIASES

Slice names identify a sequence of elements of an array object. The direction must be consistent with the direction of the identifier's array type. An ***alias*** creates a new name for all or part of the range of an array object.

Example :

variable A1 : BIT_VECTOR (7 downto 0) ;

A2 := A1(5 downto 2)

alias A3 : BIT_VECTOR (0 to 3) is A1(7 downto 4) ; (which means : A3(0)=A1(7), A3(1)=A1(6), A3(2)=A1(5), A3(3)=A1(4))

alias A4 : BIT is A1(3) ;

ATTRIBUTE NAMES

An ***attribute*** takes a variable or signal of a given type and returns a value. The following are some commonly used predefined attributes :

- **left** : returns the index of the leftmost element of the data type.
- **right** : returns the index of the rightmost element of the data type.
- **high** : returns the index of the highest element of the data type.
- **low** : returns the index of the lowest element of the data type.
- **range** : determines the index range.
- **reverse_range** : determines the index reverse_range.
- **length** : returns the number of elements of a bit_vector.
- **event** : represents whether there is a change in the signal value at the current simulation time (*associated with signals*).

ATTRIBUTE NAMES

Example :

variable A1 : BIT_VECTOR (10 downto 0) ;

A1'left returns 10

A1'right returns 0

A1'high returns 10

A1'low returns 0

A1'range returns 10 downto 0

A1'reverse_range returns 0 to 10

A1'length returns 11

AGGREGATES

An ***aggregate*** can be used to assign values to an object of array type or record type during the initial declaration or in an assignment statement.

Example :

type color_list is (red, orange, blue, white) ;
type color_array is array (color_list) of BIT_VECTOR (1 downto 0) ;
variable X : color_array ;
X := (“00”, “01”, “10”, “11”) ;
X := (red => “00”, blue => “01”, orange => “10”, white => “11”) ;

*In the second line we define an array whose element number (**index-range**) is given by **color_list**. Since **color_list** includes 4 elements, **color_array** type also includes 4 elements all of which are bit_vector. Instead of **color_list** we may use (**range 0 to 3**) , because this definition only defines the range not the element types.*

QUALIFIED EXPRESSIONS

A ***qualified expression*** states the type of the operand. The syntax is :

type_name'(*expression*)

Example :

```
type color1 is (red, orange, blue, white) ;
type color2 is (purple, green, red, brown, black) ;
```

red appears in both data_types, so it may be necessary to identify its data_type clearly as follows :

```
color2'(red)
```

TYPE CONVERSIONS

A type conversion provides for explicit conversion between closely related types. The syntax is :

type_name(*expression*)

Example :

```
signal X : STD_LOGIC_VECTOR (3 downto 0) ;
signal Y : STD_ULOGIC_VECTOR (3 downto 0) ;
Y <= STD_ULOGIC_VECTOR (X) ;
```

SEQUENTIAL STATEMENTS

Sequential statements specify the step by step behaviour of the process. They are executed starting from the first statement, then second, third until the last statement. The statements within a process are sequential statements whereas the process itself is a concurrent statement. The following are the sequential statements defined in VHDL :

- **VARIABLE** assignment statements
- **SIGNAL** assignment statements
- **IF** statements
- **CASE** statements
- **NULL** statements
- **ASSERTION** statements
- **LOOP** statements
- **NEXT** statements

SEQUENTIAL STATEMENTS

- **EXIT** statements
- **WAIT** statements
- **PROCEDURE** calls
- **RETURN** statements

VARIABLE ASSIGNMENT STATEMENTS

A ***variable assignment statement*** replaces the current value of a variable with a new value specified by an expression. The variable and the result of the expression must be of the same type. The syntax is :

target_variable := *expression* ;

When a variable is assigned, the assignment executes in zero simulation time. In other words, it changes the value of the variable immediately at the current simulation time. Variables can only be declared in a process or subprogram.

Variables declared within a process cannot pass values outside of the process ; that is they are local to a process or subprogram !!!

VARIABLE ASSIGNMENT STATEMENTS

Example :

```
subtype INT16 is INTEGER range 0 to 65535 ;
signal S1, S2 : INT16 ;
signal GT : BOOLEAN ;
process (S1, S2)
        variable A, B : INT16 ;
        constant C : INT16 :=100 ;
begin
        A := S1 + 1;
        B := S2* 2 - C ;
        GT <= A > B ;
end process ;
```

SIGNAL ASSIGNMENT STATEMENTS

A ***signal assignment statement*** replaces the current value of a signal with a new value specified by an expression. The signal and the result of the expression must be of the same type. The syntax is :

target_signal <= [**transport**] *expression* [**after** *time_expression*] ;

when a signal is assigned, the assignment will not take effect immediately, instead will be scheduled to a future simulation time. There are two types of delay that can be applied when scheduling signal assignments :

- Transport delay
- Inertial delay

TRANSPORT DELAY

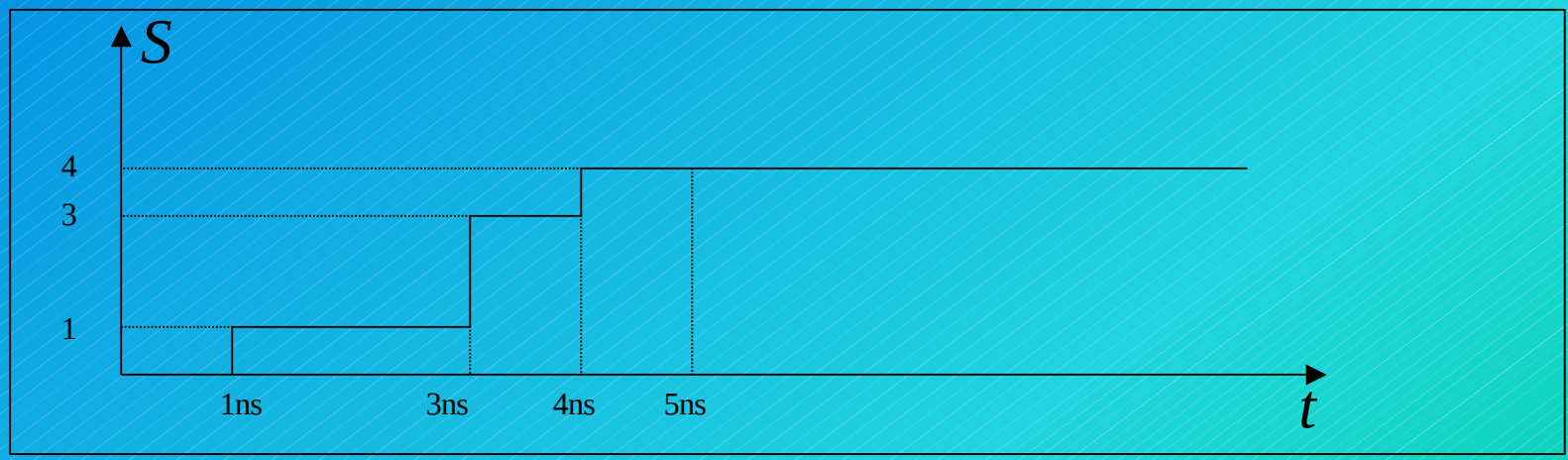
Transport delay is analogous to the delay incurred by passing a current through a wire.

If the delay time implies a transaction that follows (in time) already scheduled transactions, the new transaction is added to the end of all the others.

If the delay time implies a transaction that precedes (in time) already scheduled transactions, the new transaction overrides all the others.

TRANSPORT DELAY

```
…….
    Process (…….)
    begin
        S <= transport 1 after 1 ns, 3 after 3 ns, 5 after 5 ns ;
        S <= transport 4 after 4 ns ;
    end ;
…….
```



INERTIAL DELAY

Inertial delay is the default in VHDL. It is used for devices that do not respond unless a value on its input persists for the given amount of time. It is useful in order to ignore input glitches whose duration is less than the port delay.

If the delay time implies a transaction that follows (in time) and is different from (in value) the transactions already scheduled by other statements, the new transaction overrides the others. If the value is the same the new transaction is added to the end.

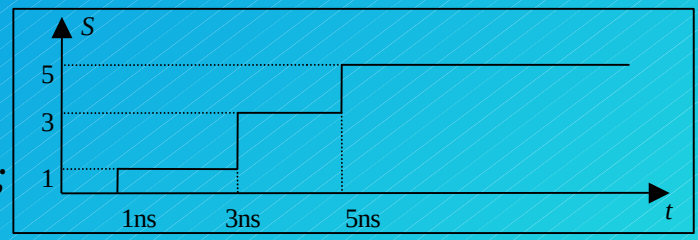
If the delay time implies a transaction that precedes (in time) already scheduled transactions, the new transaction overrides all the others.

The second assignment always overrides the first assignment !!!

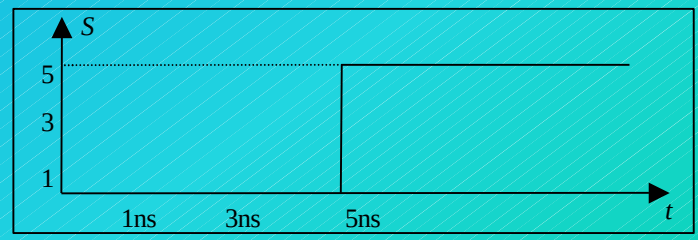
INERTIAL DELAY

Examples :

```
……..
    Process (…….)
    begin
        S <= 1 after 1 ns, 3 after 3 ns, 5 after 5 ns ;
    end ;
……..
```

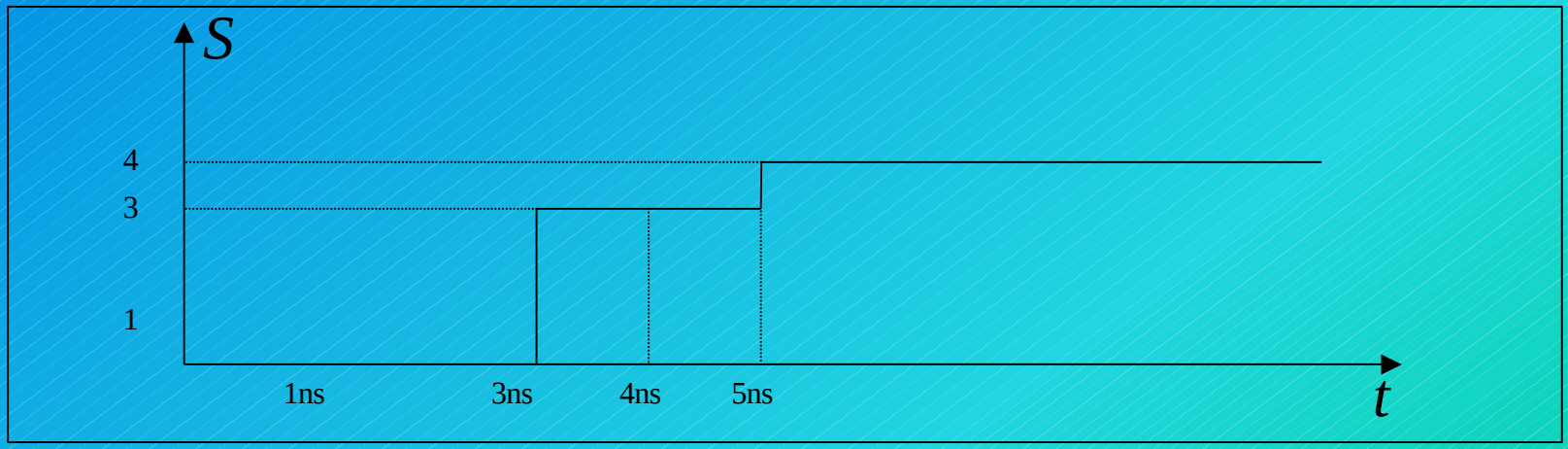



```
……..
    Process (…….)
    begin
        S <= 1 after 1 ns ;
        S <= 3 after 5 ns ;
        S <= 5 after 5 ns ;
    end ;
……..
```



INERTIAL DELAY

```
……..
    Process (…….)
    begin
        S <= 1 after 1 ns, 3 after 3 ns, 5 after 5 ns, 6 after 6 ns ;
        S <= 3 after 4 ns, 4 after 5 ns ;
    end ;
……..
```



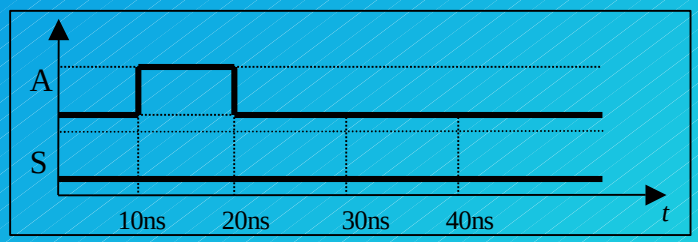
INERTIAL DELAY versus TRANSPORT DELAY

In an ***inertial*** model, glitches (on the input signal) with a duration which is less than the delay through the device will not be present on the output signal. In a ***transport*** model, glitches (on the input signal) of any duration will be always present on the output signal.

Example : let's assign the value of signal A to signal S with a 20 ns delay time. If in signal A, a pulse with 10 ns duration occurs at time t=10 ns then we will have the following situation :

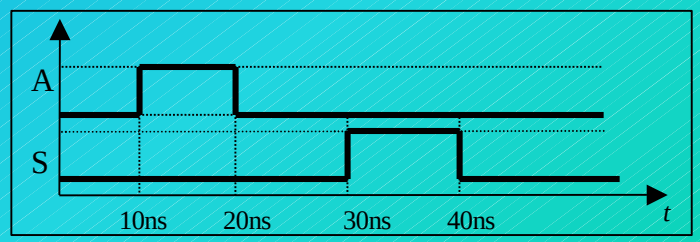
INERTIAL *case*

S <= A after 20 ns ;



TRANSPORT *case*

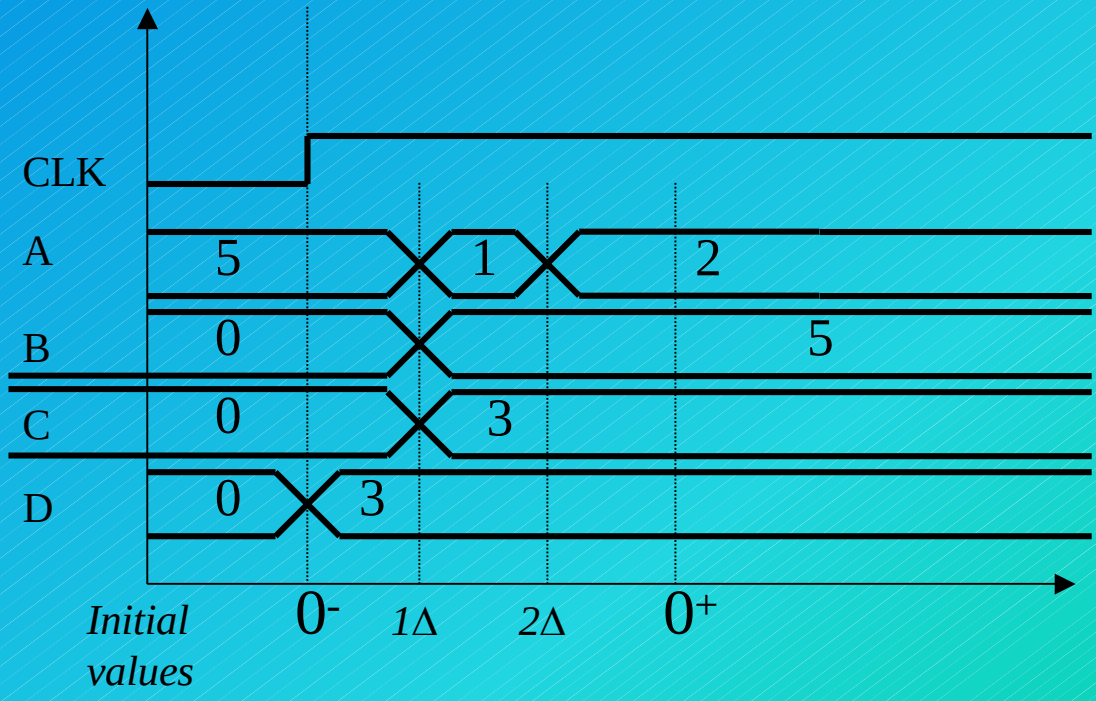
S <= transport A after 20 ns ;



ZERO DELAY and DELTA DELAY

Variable assignments are executed in zero time. However VHDL uses delta time concept for signal assignments. Each signal assignment statement is executed after a delta time.

```
process (CLK)
signal A : integer := 5 ;
       B, C : integer := 0 ;
variable D : integer := 0 ;
begin
       A <= 1;
       A <= 2;
       B <= A;
       D := 3;
       C <= D;
end process ;
```



ZERO DELAY and DELTA DELAY

The process is activated by any change in the CLK signal. The CLK changes in zero time. 0^- and 0^+ are both 0 for a simulator. The interval, two delta (2Δ) is a virtual concept. A signal assignment is executed after a delta delay however variable assignments are executed in zero time. The first assignment is a signal assignment, therefore ***A*** will be assigned ***"1"*** after a delta time. The second assignment is also a signal assignment so ***A*** will be ***"2"*** after two delta time. Third assignment assigns signal ***B***, the initial value of ***A*** (the value at 0^- time) because delta time concept is virtual. So ***B*** takes ***"5"*** after a delta time. Fourth assignment is a variable assignment, so it will be executed without delta delay. The last assignment is again a signal assignment ; signal ***C*** takes the value of ***D*** after a delta time. Since ***D*** is ***"3"*** at zero time ***C*** is assigned to ***"3".***

This is why signal assignments should be avoided in processes. If we define signal A as a variable B takes the value of ***"2"*** .

IF STATEMENTS

A ***if statement*** selects for execution one or more of the enclosed sequences or statements, depending upon the value of one or more corresponding conditions. The syntax is :

if *condition* **then**
    {*sequential_statement*}
{**elsif** *condition* **then**
    {*sequential_statement*}}
[**else**
    {*sequential_statement*}]
end if ;

IF STATEMENTS

An expression specifying a ***condition*** must be a BOOLEAN type expression. The ***condition*** of the ***if*** statement is first evaluated. If the ***condition*** is TRUE, then the statement immediately following the keyword "***then***" is executed ; else the conditions following the ***elsif clauses*** are evaluated step by step.

The final ***else*** is treated as "***elsif TRUE then***", so if none of the conditions before ***else clause*** are TRUE , then ***else statements*** will be executed.

IF STATEMENTS

Example :

```
signal IN1, IN2, OU : STD_LOGIC ;
process (IN1, IN2)
begin
        if IN1 ='0' or IN2='0' then
                OU <='0';
        elsif IN1='X' or IN2='X' then
                OU <='1';
        else
                OU <='1';
        end if ;
end process ;
```

CASE STATEMENTS

The ***case statement*** selects, for execution, one of a number of alternative sequences of statements. The chosen alternative is defined by the value of an expression. The syntax is :

case *expression* **is**
when *choices* =>
{*sequential_statement*}}
{**when** *choices* =>
{*sequential_statement*}}
end case ;

Each choice must be of the same type as the ***expression***. Each value must be represented once and only once in the set of choices of the case statement. If no ***others*** choice is presented, all possible values of the expression must be covered by the set of choices.

CASE STATEMENTS

Example :

```
signal S1 : INTEGER range 0 to 7 ;
signal I1, I2, I3 : BIT ;
process (S1, I1, I2, I3)
begin
        case S1 is
                when 0 | 2 =>
                        OU <= '0' ;
                when 1 =>
                        OU <= I1 ;
                when 3 to 5 =>
                        OU <= I2 ;
                when others =>
                        OU <= I3 ;
        end case ;
end process ;
```

NULL STATEMENTS

There is no action for a ***null statement*** in VHDL. The system will ignore the null statement and proceed to the next statement. This statement is usually used to explicitly state that no action is to be performed when a condition is true. The syntax is :

null ;

Example :

```
variable A, B : INTEGER range 0 to 31 ;
case A is
        when 0 to 12 =>
                B := A ;
        when others =>
                null ;
end case ;
```

ASSERTION STATEMENTS

During simulation, it is convenient to output a text string message as a warning or error message. The ***assert statement*** allows for testing a condition and issuing a message. The ***assert statement*** checks to determine if a specified condition is true, and displays a message if the condition is false. The syntax is :

assert *condition* [**report** *error_message*]
[**severity** *severity_expression*];

where the ***condition*** must be a BOOLEAN type. The ***error message*** is a STRING type expression and the ***severity expression*** is of predefined type SEVERITY_LEVEL. There are four levels of severity : FAILURE, ERROR, WARNING, NOTE. The severity level is used (in the simulator) either to terminate a simulation run or just to give a warning message and continue.

ASSERTION STATEMENTS

The assert statement is useful for timing checks, out-of-range conditions, etc.

Example :

```
assert (X >3)                    (prints if condition is false !!!)
report "setup violation"
severity warning ;
```

To unconditionally print out a message, use the condition ***false***.

```
assert (false)
report "starting simulation" ;
```

LOOP STATEMENTS

A ***loop statement*** include a sequence of statements to be executed repeatedly, zero or more times. The syntax is :

[*label* :] [**while** *condition* | **for** *loop_specification*] **loop**
{*sequential_statements*} |
{**next** [*label*] [**when** *condition*] ;} |
{**exit** [*label*] [**when** *condition*] ;}
end loop [*label*];

There are two different styles of the loop statement : **FOR LOOP** and **WHILE LOOP**. These are called ***iteration schemes***. You can also define a loop without an iteration scheme which means repeated execution of the statements. However in such a case you have to use a ***wait statement*** and an ***exit statement***.

LOOP STATEMENTS

Example : *The following example shows two nested loops without an iteration scheme (**for** or **while**).*

```
count_down : process
      variable min, sec : integer range 0 to 60 ;
begin
      l1 : loop
            l2 : loop
                  exit l2 when (sec = 0) ;
                  wait until CLK'event and CLK ='1' ;
                  sec := sec -1 ;
            end loop l2 ;
            exit l1 when (min = 0) ;
            min := min - 1;
            sec := 60 ;
      end loop l1 ;
end process count_down ;
```

FOR LOOP STATEMENTS

A ***for loop*** is a sequential statement in a process that iterates over a number of values. The ***loop index*** does not have to be declared, and it can be reassigned a value within the loop. It is by default integer.

Example :

```
for i in 1 to 10 loop
        a(i) := i * i ;
end loop ;

for I in X downto Y loop
        a(i) := i * i ;
end loop ;
```

WHILE LOOP STATEMENTS

A ***while loop*** executes the loop body by first evaluating the ***condition***. If the condition is TRUE, then the loop is executed.

Example :

```
process
    variable a, b, c, d : integer ;
begin
    ……..
    while ((a + b) > (c+d)) loop
        a := a-1 ;
        c := c+b ;
        b := b-d ;
    end loop ;
    ……….
end process ;
```

NEXT STATEMENTS

The ***next statement*** skips execution to the next iteration of an enclosing loop statement (*called* ***loop_label*** *in the syntax*). If the loop_label is absent, the next statement applies to the innermost enclosing loop. The syntax is :

next [*loop_label*] [**when** *condition*] ;

Example :

```
l1 :  while a < 10  loop
        l2 : while b < 20 loop
                .
                next l1 when a = b ;
                .
        end loop l2 ;
end loop l1 ;
```

EXIT STATEMENTS

The ***exit statement*** completes the execution of an enclosing LOOP statement (*called* ***loop_label*** *in the syntax*) and continues with the next statement after the exited loop. If the loop_label is absent, the exit statement applies to the innermost enclosing loop. The syntax is :

exit [*loop_label*] [**when** *condition*] ;

Example :

```
for a in 0 to 10 loop
    exit when X(a) = 0 ;
    Y(a) := X(a) ;
end loop ;
```

WAIT STATEMENTS

The ***exit statement*** causes a simulator to suspend execution of a process statement or a subprogram, until some conditions are met. The objects being waited upon should be signals. The syntax is :

wait

[**on** *signal_name* {, *signal_name*}]

[**until** *boolean_expression*]

[**for** *time_expression*] ;

Example :

wait on a, b ;

wait until x < 10 ;

wait for 10 us ;

wait on a,b until (x < 10) for 10 us ;

wait until (CLK'event and CLK = '1') ; **(waits for the rising edge of the CLK!!!)**

PROCEDURE CALLS

In a behaviour design description, subprograms provide a convenient way of documenting frequently used functions. There are two different types of subprograms :

A ***procedure*** (*returns multiple values*) and a ***function*** (*returns a single value*). A subprogram is composed of sequential statements just like a process.

Procedure calls invoke procedures to be executed in a process.

RETURN STATEMENTS

The return statement terminates a subprogram. The return statement can only be described within a function or a procedure. It is required in function body but optional in a procedure body. The syntax is :

return [*expression*] ;

where expression provides the function's return value. The return statement within a function must have an expression as its return value, but the return statement appeared in procedures must not have the expression. A function can have more than one return statement. But only one return statement is reached by a given function call.

CONCURRENT STATEMENTS

Concurrent statements are executed in parallel at the same simulated time. It does not matter on the order they appear in the architecture. Concurrent statements pass information through signals. The following are the concurrent statements defined in VHDL :

- **PROCESS** assignment statements
- **Concurrent SIGNAL** assignment statements
- **Conditional SIGNAL** assignment statements
- **Selected SIGNAL** assignment statements
- **BLOCK** statements
- **Concurrent PROCEDURE** calls
- **Concurrent ASSERTION** statements

PROCESS STATEMENTS

A ***process*** is composed of a set of sequential statements, but processes are themselves concurrent statements. All the processes in a design execute concurrently. However, at any given time only one sequential statement is executed within each process. A process communicates with the rest of a design by reading or writing values to and from signals or ports declared outside the process. The syntax is :

[*label* :] **process** [(*sensitivity_list*)]
{*process_declaration_part*}
begin
{*sequential_statements*}
end process [*label*] ;

PROCESS STATEMENTS

A ***process_declaration_part*** defines objects that are local to the process, and can have any of the following items :

- variable declaration
- constant declaration
- type declaration
- subtype declaration
- subprogram body
- alias declaration
- use clause

We have already described ***sequential_statements***.

PROCESS STATEMENTS

A ***sensitivity list*** has the same meaning as a process containing a wait statement as the last statement and interpreted as ;

wait on *sensitivity_list* ;

The process is like an infinite loop statement which encloses the whole sequential statements specified in the process. Therefore ;

The process statement must have either a sensitivity list or a wait statement (or both) !!!

PROCESS STATEMENTS

Example :

```
architecture A2 of example is
signal i1, i2, i3, i4, and_out, or_out : bit ;
begin
        pr1 : process(i1, i2, i3, i4)
        begin
                and_out <= i1 and i2 and i3 and i4 ;
        end process pr1;
        pr2 : process(i1, i2, i3, i4)
        begin
                or_out <= i1 or i2 or i3 or i4 ;
        end process pr2;
end A2 ;
```

CONCURRENT SIGNAL ASSIGNMENTS

Another form of a signal assignment is a ***concurrent signal assignment***, which is used outside of a process, but within an architecture. The syntax is :

target_signal <= *expression* [**after** *time_expression*] ;

Similar to the sequential signal assignment, ***the after clause is ignored by the synthesizer !!!***

Any signal on the right side of the assignment is like a sensitivity list element.

CONCURRENT SIGNAL ASSIGNMENTS

Example : *All the examples are equivalent.*

```
architecture A1 of example is
signal i1, i2, i3, i4, and_out, or_out : bit ;
begin
        and_out <= i1 and i2 and i3 and i4 ;
        or_out <= i1 or i2 or i3 or i4 ;
end A1 ;
```

```
architecture A3 of example is
signal i1, i2, i3, i4, and_out, or_out : bit ;
begin
        process
        begin
                and_out <= i1 and i2 and i3 and i4 ;
                or_out <= i1 or i2 or i3 or i4 ;
                wait on i1, i2, i3, i4 ;
        end process ;

end A3 ;
```

```
architecture A2 of example is
signal i1, i2, i3, i4, and_out, or_out : bit ;
begin
        process(i1, i2, i3, i4)
        begin
                and_out <= i1 and i2 and i3 and i4 ;
        end process ;
        process(i1, i2, i3, i4)
        begin
                or_out <= i1 or i2 or i3 or i4 ;
        end process ;
end A2 ;
```

CONDITIONAL SIGNAL ASSIGNMENTS

A ***conditional signal assignment*** is a concurrent statement and has one target, but can have more than one expression. Except for the final expression, each expression goes with a certain condition. The conditions are evaluated sequentially. If one condition evaluates to TRUE, then the corresponding expression is used ; otherwise the remaining expression is used. One and only one expression is used at a time. The syntax is :

target <= {*expression* [**after** *time_expression*] **when** *condition* **else**}
expression [**after** *time_expression*] ;

Any conditional signal assignment can be described by a process statement which contains an ***if statement***.

You cannot use conditional signal assignments in a process !!!

CONDITIONAL SIGNAL ASSIGNMENTS

Example : *The examples are equivalent.*

```
architecture A1 of example is
signal a, b, c, d : integer ;
begin
        a <= b when (d > 10) else
                c when (d > 5) else
                d ;
end A1 ;
```

```
architecture A2 of example is
signal a, b, c, d : integer ;
begin
        process(b, c, d)
        begin
                if (d > 10) then
                        a <= b ;
                elsif (d > 5) then
                        a <= c ;
                else
                        a <= d ;
                end if ;
        end process ;
end A2 ;
```

SELECTED SIGNAL ASSIGNMENTS

A ***selective signal assignment*** can have only one target and can have only one ***with*** expression. This value is tested for a match in a manner similar to the ***case*** statement. It runs whenever any change occurs to the selected signal. The syntax is :

with *choice_expression* **select**

target <= {*expression* [**after** *time_expression*] **when** *choices* ,}

expression [**after** *time_expression*] **when** *choices* ;

Any selected signal assignment can be described by a process statement which contains a ***case statement***.

You cannot use selected signal assignments in a process !!!

SELECTED SIGNAL ASSIGNMENTS

Example : *The examples are equivalent.*

```
with SEL select
    z <=   a when 0 | 1 | 2 ,
           b when 3 to 10 ,
           c when others ;
```

```
process (SEL, a, b, c)
begin
      case SEL is
            when 0 | 1 | 2 | =>
                  z <= a ;
            when 3 to 10 =>
                  z <= b ;
            when others =>
                  z <= c ;
      end case ;
end process ;
```

BLOCK STATEMENTS

Blocks allow the designer to logically group sections of a concurrent model, sections that are not scheduled to be used in other models (and that's when blocks are used instead of components). Blocks are used to organise a set of concurrent statements hierarchically. The syntax is :

label : **block**
    {*block_declarative_part*}
begin
    {*concurrent_statement*}
end block [*label*] ;

A block declarative part defines objects that are local to the block, and can have any of the following items

BLOCK STATEMENTS

A ***block declarative part*** defines objects that are local to the block, and can have any of the following items :

- signal declaration
- constant declaration
- type declaration
- subtype declaration
- subprogram body
- alias declaration
- use clause
- component declaration

Objects declared in a block are visible to that block and all blocks nested within. When a child block declares an object with the same name as the one in the parent block, child's declaration overrides the parent's.

BLOCK STATEMENTS

Example : *In the next example, block B1-1 is nested within block B1. Both B1 and B1-1 declare a signal named S. the signal S used in the block B1-1 will be the one declared within block B1-1, while the S used in block B2 is the one declared in B1.*

```
architecture BHV of example is
        signal out1 : integer ;
        signal out2 : bit ;
begin
        B1 : block
                signal S : bit ;
        begin
                B1-1 : block
                        signal S : integer ;
                begin
                        out1 <= S ;
                end block B1-1 ;
        end block B1 ;
        B2 : block
        begin
                out2 <= S ;
        end block B2 ;
end BHV ;
```

CONCURRENT PROCEDURE CALLS

A concurrent procedure call is a procedure call that is executed outside of a process ; it stands alone in an architecture. Concurrent procedure call :

- Has IN, OUT, and INOUT parameters.
- May have more than one return value.
- Is considered a statement.
- Is equivalent to a process containing a single procedure call.

CONCURRENT PROCEDURE CALLS

Example : *The examples are equivalent.*

```
architecture ……
begin
        procedure_any (a, b) ;
end  ….. ;
```

```
architecture ……
begin
        process
        begin
                procedure_any (a, b) ;
                wait on a, b ;
        end process ;
end  ….. ;
```

CONCURRENT ASSERTION STATEMENTS

The concurrent assertion statement performs the same action and is used for the same reason as the sequential assertion statements within a process.

This statement is used for simulation purpose only and will be ignored by the synthesis tool !!!

SUBPROGRAMS

Subprograms consist of procedures and functions that can be invoked repeatedly from different locations in a VHDL description. VHDL provides two kinds of subprograms :

- procedures
- functions

FUNCTIONS

Functions :

- Are invoked as expressions.
- Always return ***just one*** argument.
- All parameters of functions must be of mode ***in***.
- All parameters of functions must be class of ***signal*** or ***constant***.
- Must declare the ***type*** of the value it returns.
- Cannot contain ***wait statements***.

FUNCTIONS

The syntax is :

function *identifier interface_list* **return** *type_mark* **is**
    {*subprogram_declarative_item*}
begin
    {*sequential_statement*}
end [*identifier*] ;

The ***identifier*** defines the name of the function, and the ***interface_list*** defines the formal parameters of the function. Each parameter is defined using the following syntax :

FUNCTIONS

[class] *name_list* [mode] *type_name* [:= *expression*] ;

where the ***class*** of a object refers to ***constant*** or ***signal*** and the ***mode*** of a object must be ***in***. If no mode is specified the parameter is interpreted as mode ***in.*** If no class is specified, parameters are interpreted as class ***constant***.

Example :

```
process
                function c_to_f (c : real) return real is
                                variable f : real ;
                begin
                                f : c * 9.0 / 5.0 + 32.0 ;
                                return ( f ) ;
                end c_to_f ;

                variable temp : real ;
begin
                temp := c_to_f ( 5.0 ) + 20.0 ; (temp = 61)
end process ;
```

By default will be understood as :

constant c : in real ;

PROCEDURES

Procedures :

- Are invoked as statements..
- Can return ***none***, ***one*** or ***more*** argument.
- Parameters of procedures may be of mode ***in***, ***out*** and ***inout***.
- All parameters must be class of ***signal***, ***constant*** or ***variable***.
- May contain ***wait statements***.

PROCEDURES

The syntax is :

procedure *identifier interface_list* **is**
    {*subprogram_declarative_item*}
begin
    {*sequential_statement*}
end [*identifier*] ;

The ***identifier*** defines the name of the procedure, and the ***interface_list*** defines the formal parameters of the procedure. Each parameter is defined using the following syntax :

PROCEDURES

[class] *name_list* [mode] *type_name* [:= *expression*] ;

where the ***class*** of a object refers to ***constant***, ***variable*** or ***signal*** and the ***mode*** of a object may be ***in***, ***out*** or ***inout***. If no mode is specified the parameter is interpreted as mode ***in.*** If no class is specified, parameters of mode ***in*** are interpreted as class ***constant***, and parameters of mode ***out*** and ***inout*** are interpreted as being of class ***variable***.

PROCEDURES

Example :

```
procedure parity (A : in bit_vector (0 to 7) ;
                  result1, result2 : out bit) is
    variable temp : bit ;
    begin
        temp := '0' ;
        for I in 0 to 7 loop
            temp := temp xor A(I) ;
        end loop ;
        result1 := temp ;
        result2 := not temp ;
    end ;
```

By default will be understood as :

variable result1, result2 : out bit ;

```
architecture BHV of receiver is
begin
  process
    variable TOP, BOTTOM, ODD, dummy : bit ;
    variable y : bit_vector ( 15 downto 0) ;
  begin
    .
    .
    parity ( y(15 downto 8) , TOP, dummy) ;
    parity ( y(7 downto 0) , BOTTOM, dummy) ;
    ODD := TOP xor BOTTOM ;
  end process ;
end BHV;
```

Procedure call :

parity (x, y, z); **(variable x, y : bit ; NOT signal x, y : bit;)**

parity (A => x, result1 => y, result2 => z) ;

PACKAGES

You create a ***package*** to store common subprograms, data types, constants, etc. that you can use in more than one design. A package consists of two parts : a ***package declaration section*** and a ***package body***. The package declaration defines the interface for the package. The syntax is :

package *package_name* **is**
    {*package_declarative_item*}
end [*package_name*] ;

PACKAGES

The ***package_declarative_item*** can be any of these :

- type declaration
- subtype declaration
- signal declaration
- constant declaration
- alias declaration
- component declaration
- subprogram declaration
- use clause (*to include other packages*)

Signal declarations in a package pose some problems in synthesis because a signal cannot be shared by two entities. A common solution is to make it a global signal !!!

PACKAGES

The package body specifies the actual behaviour of the package. A package body always has the same name as its corresponding package declaration. The syntax is :

package body *package_name* **is**

{*package_body_declarative_item*}

end [*package_name*] ;

The ***package_body_declarative_item*** can be any of these :

- type declaration
- subtype declaration
- constant declaration
- use clause
- subprogram body

PACKAGES

```
library IEEE ;
use IEEE.NUMERIC_BIT.all ;
package PKG is
        subtype MONTH_TYPE is integer range 0 to 12 ;
        subtype DAY_TYPE is INTEGER range 0 to 31 ;
        subtype BCD4_TYPE is unsigned (3 downto 0) ;
        subtype BCD5_TYPE is unsigned (4 downto 0) ;
        constant BCD5_1 : BCD5_TYPE := B"0_0001" ;
        constant BCD5_7 : BCD5_TYPE := B"0_0111" ;
        function BCD_INC ( L : in BCD4_TYPE) return BCD5_TYPE ;
end PKG ;

package body PKG is
        function BCD_INC ( L : in BCD4_TYPE) return BCD5_TYPE is
                variable V, V1, V2 : BCD5_TYPE ;
        begin
                V1 := L + BCD5_1 ;
                V2 := L + BCD5_7 ;
                case V2(4) is
                        when '0' => V := V1 ;
                        when '1' => V := V2 ;
                end case ;
                return (V) ;
        end BCD_INC ;
end PKG ;
```

MODELLING AT THE STRUCTURAL LEVEL

A digital system is usually represented as a hierarchical collection of components. Each component has a set of ports which communicate with the other components. In a VHDL description, a design hierarchy is introduced through component declarations and component instantiation statements.

While the basic unit of a behaviour description is the process statement, the basic unit of a structural description is the component instantiation statement.

Both the process statements and the and the component instantiation statements must be enclosed in an architecture body.

COMPONENT DECLARATIONS

An architecture body can use other entities described separately and placed in the design libraries using ***component declaration*** and ***component instantiation*** statements. In a design description, each component declaration statement corresponds to an entity. The component declaration statement is similar to the entity specification statement in that it defines the component's interface. A component declaration is required to make a design entity useable within the current design. The syntax is :

component *component_name*
[**port** (*local_port_declarations*)]
end component ;

component_name represents the name of the entity, and ***port_declarations*** are the same as that defined for entity declaration.

COMPONENT INSTANTIATIONS

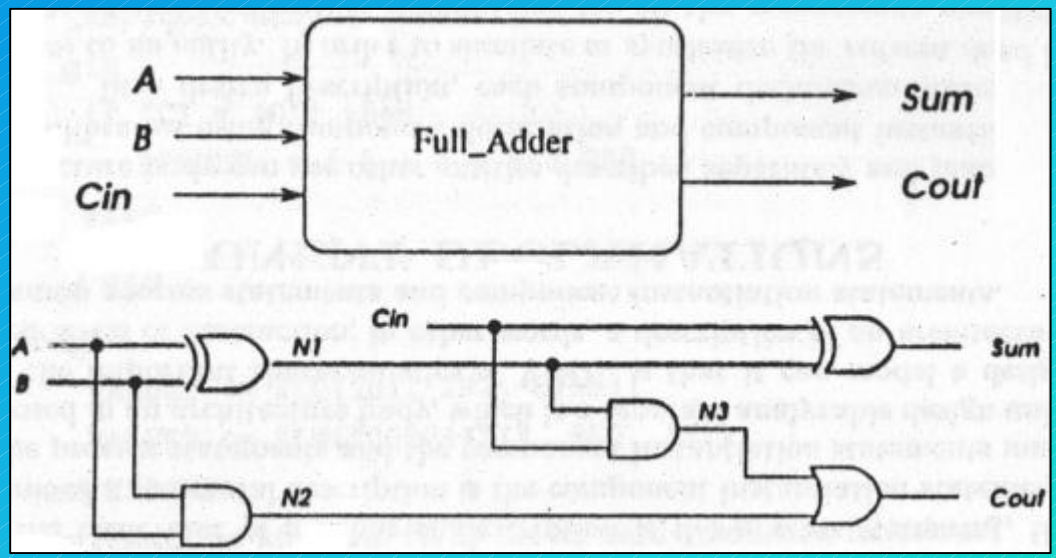
A component defined in an architecture may be instantiated using ***component instantiation*** statement. At the point of instantiation, only the external view of the component (the name, type, direction of its ports) is visible, signals internal to the component are not visible. The syntax is :

instantiation_label : *component_name*
port map (
[*local_port_name* =>] *expression*
{, [*local_port_name* =>] *expression*}
) ;

A component instantiation statement must be preceded by an ***instantiation_label***.

COMPONENT INSTANTIATIONS

Figure shows the interface and also the implementation of a full adder. In this implementation, three types of components : ***OR2_gate***, ***AND2_gate***, and ***XOR_gate*** are used to build the full adder circuit. The following shows the entity and architecture specifications of the components used in the full adder design.



COMPONENT INSTANTIATIONS

```
library IEEE ;
use IEEE.STD_LOGIC_1164.all ;
entity AND2_gate is
        port ( I0, I1 : in STD_LOGIC ;  O : out STD_LOGIC );
end AND2_gate ;
architecture BHV of AND2_gate is
begin
        O <= I0 and I1 ;
end BHV ;
```

```
library IEEE ;
use IEEE.STD_LOGIC_1164.all ;
entity OR2_gate is
        port ( I0, I1 : in STD_LOGIC ;
                O : out STD_LOGIC );
end OR2_gate ;
architecture BHV of OR2_gate is
begin
        O <= I0 or I1 ;
end BHV ;
```

```
library IEEE ;
use IEEE.STD_LOGIC_1164.all ;
entity XOR_gate is
        port ( I0, I1 : in STD_LOGIC ;  O : out STD_LOGIC );
end XOR_gate ;
architecture BHV of XOR_gate is
begin
        O <= I0 xor I1 ;
end BHV ;
```

COMPONENT INSTANTIATIONS

```
library IEEE ;
use IEEE.STD_LOGIC_1164.all ;
entity FULL_ADDER is
        port ( A, B, Cin : in STD_LOGIC ;
                Sum, Cout : out STD_LOGIC );
end FULL_ADDER ;
architecture IMP of FULL_ADDER is
        component XOR_gate
                port ( I0, I1 : in STD_LOGIC ;  O : out STD_LOGIC );
        end component ;
        component AND2_gate
                port ( I0, I1 : in STD_LOGIC ;  O : out STD_LOGIC );
        end component ;
        component OR2_gate
                port ( I0, I1 : in STD_LOGIC ;  O : out STD_LOGIC );
        end component ;
        signal N1, N2, N3 : STD_LOGIC ;
begin
        U1 : XOR_gate port map (I0 => A, I1 => B, O => N1 ) ;
        U2 : AND2_gate port map ( A, B, N2 ) ;
        U3 : AND2_gate port map ( Cin, N1, N3 ) ;
        U4 : XOR_gate port map ( Cin, N1, Sum ) ;
        U5 : OR2_gate port map ( N3, N2, Cout ) ;
end IMP ;
```

GENERATE STATEMENTS

The ***generate statement*** is a concurrent statement that has to be defined in an architecture. It is used to describe replicated structures. The syntax is :

instantiation_label : *generation_scheme* **generate**

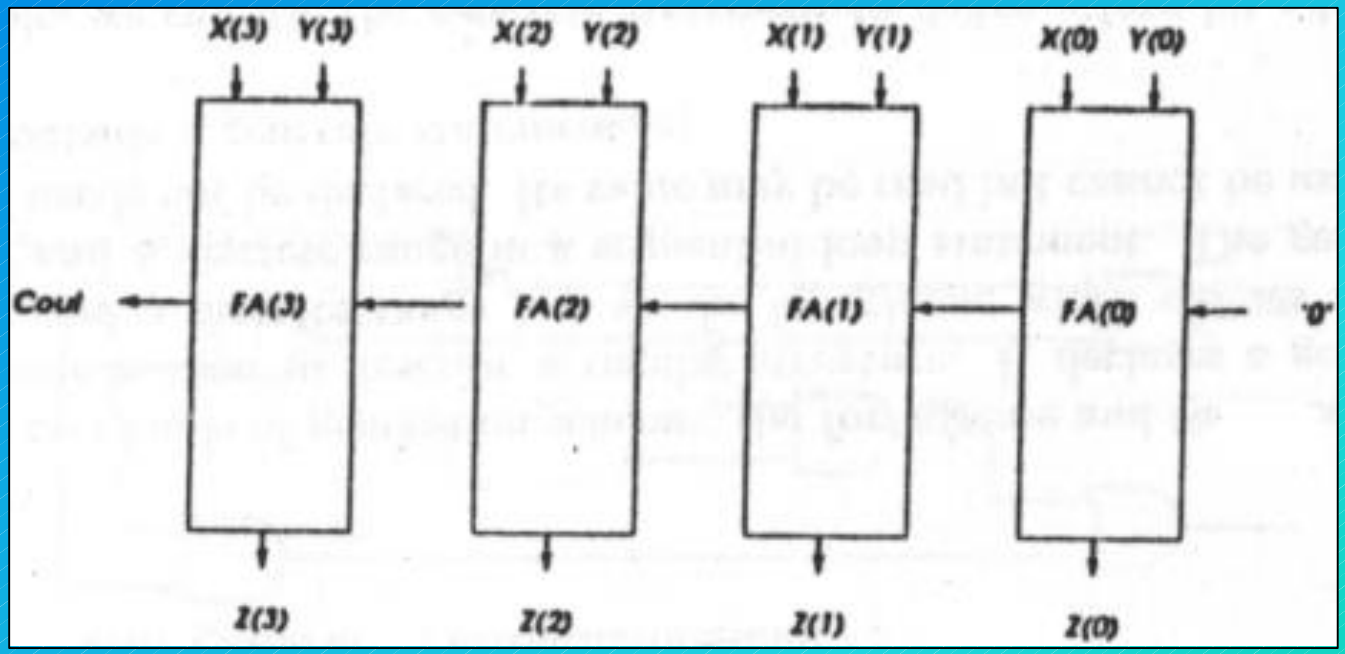
{*concurrent_statement*}

end generate [*instantiation_label*] ;

There are two kinds of ***generation_scheme*** : the ***for*** scheme and the ***if*** scheme. A for scheme is used to describe a regular structure. It declares a generate parameter and a discrete range just as the for_scheme which defines a loop parameter and a discrete range in a sequential loop statement. ***The generate parameter needs not to be declared***. Its value may be read but cannot be assigned or passed outside a generate statement.

GENERATE STATEMENTS

Figure shows a four-bit adder which includes four FULL_ADDER components.

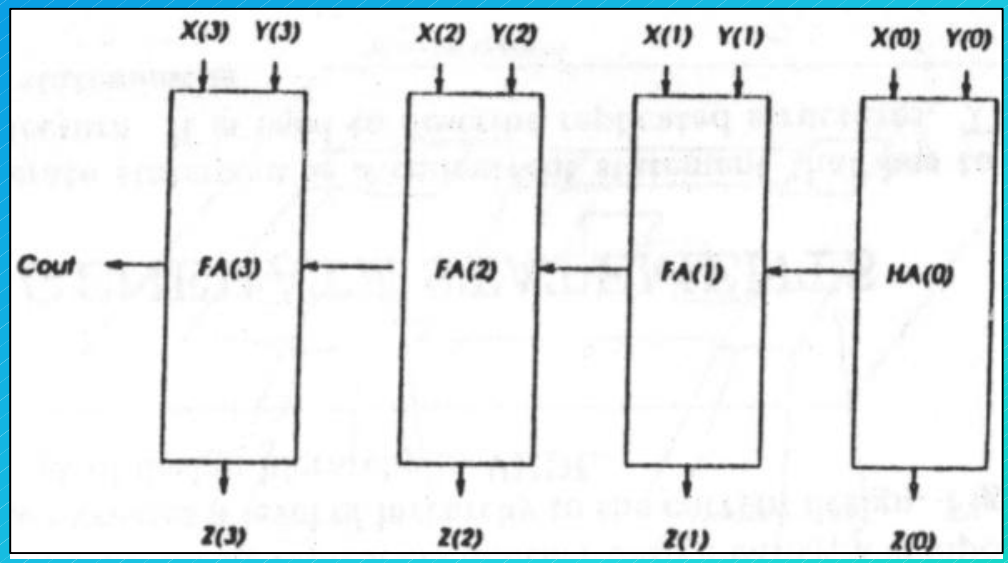


GENERATE STATEMENTS

```
architecture IMP of FULL_ADDER4 is
        signal X, Y, Z : STD_LOGIC_VECTOR ( 3 downto 0 ) ;
        signal Cout : STD_LOGIC ;
        signal TMP : STD_LOGIC_VECTOR ( 4 downto 0 ) ;
        component FULL_ADDER
                port ( A, B, Cin : in STD_LOGIC ;
                        Sum, Cout : out STD_LOGIC );
        end component ;
begin
        TMP( 0 ) <= '0' ;
        G : for I in 0  to 3 generate
                FA : FULL_ADDER port map (  X( I ), Y( I  ), TMP ( I ), Z ( I ), TMP ( I + 1 ) ) ;
                end generate ;
        Cout <= TMP ( 4 ) ;
end IMP ;
```

GENERATE STATEMENTS

Many regular structures have some irregularities. An ***if scheme*** is designed for such conditions. Unlike the sequential ***if statement***, the ***if generate*** cannot have ***else*** or ***elsif*** branches. ***Figure*** shows a four-bit adder which includes three FULL_ADDER components and a HALF_ADDER component.



GENERATE STATEMENTS

```
architecture IMP of FULL_ADDER4 is
        signal X, Y, Z : STD_LOGIC_VECTOR ( 3 downto 0 ) ;
        signal Cout : STD_LOGIC ;
        signal TMP : STD_LOGIC_VECTOR ( 4 downto 1 ) ;
        component FULL_ADDER
                port ( A, B, Cin : in STD_LOGIC ;
                        Sum, Cout : out STD_LOGIC );
        end component ;
        component HALF_ADDER
                port ( A, B : in STD_LOGIC ;
                        Sum, Cout : out STD_LOGIC );
        end component ;
begin
        G0 : for I in 0  to 3 generate
                G1 : if I = 0 generate
                        HA : HALF_ADDER port map (  X( I ), Y( I  ), Z ( I ), TMP ( I + 1 ) ) ;
                end generate ;
                G2 : if I >= 1 and I <= 3 generate
                        FA : FULL_ADDER port map (  X( I ), Y( I  ), TMP ( I ), Z ( I ), TMP ( I + 1 ) ) ;
                end generate ;
        end generate ;
        Cout <= TMP ( 4 ) ;
end IMP ;
```

CONFIGURATION SPECIFICATIONS

An entity may have several architectures. Configuration specification allows the designer to choose the entities and their architectures. The syntax is :

for *instantiation_list* : *component_name*
use entity *library_name* . *entity_name* [(*architecture_name*)] ;

If there is only one architecture, the architecture_name can be omitted.

```
library IEEE ;
use IEEE.STD_LOGIC_1164.all ;
entity FULL_ADDER is
        port ( A, B, Cin : in STD_LOGIC ;
                Sum, Cout : out STD_LOGIC );
end FULL_ADDER ;
```

CONFIGURATION SPECIFICATIONS

```
architecture IMP of FULL_ADDER is
        component XOR_gate
                port ( I0, I1 : in STD_LOGIC ;  O : out STD_LOGIC );
        end component ;
        component AND2_gate
                port ( I0, I1 : in STD_LOGIC ;  O : out STD_LOGIC );
        end component ;
        component OR2_gate
                port ( I0, I1 : in STD_LOGIC ;  O : out STD_LOGIC );
        end component ;
        signal N1, N2, N3 : STD_LOGIC ;

        for U1 : XOR_gate use entity work . XOR_gate ( BHV ) ;
        for others : XOR_gate use entity work . XOR_gate ( BHV ) ;
        for all : AND2_gate use entity work . AND2_gate ;
        for U5 : OR2_gate use entity work . OR2_gate  ;
begin
        U1 : XOR_gate port map (I0 => A, I1 => B, O => N1 ) ;
        U2 : AND2_gate port map ( A, B, N2 ) ;
        U3 : AND2_gate port map ( Cin, N1, N3 ) ;
        U4 : XOR_gate port map ( Cin, N1, Sum ) ;
        U5 : OR2_gate port map ( N3, N2, Cout ) ;
end IMP ;
```

MODELLING A TEST BENCH

To test a compiled VHDL design, a ***test bench*** is needed. A ***test bench does not have external ports***. To test our FULL_ADDER we can use the following test bench :

```
library IEEE ;
use IEEE.STD_LOGIC_1164.all ;
entity test_FA is
end test_FA ;
architecture testbench of test_FA is
        component FULL_ADDER
                port ( A, B, Cin : in STD_LOGIC ;
                        Sum, Cout : out STD_LOGIC ) ;
        end component ;
        for U1 : FULL_ADDER use entity work . FULL_ADDER ( IMP ) ;
        signal A, B, Cin , Sum, Cout  : STD_LOGIC ;
begin
        U1 : FULL_ADDER port map ( A, B, Cin, Sum, Cout ) ;
```

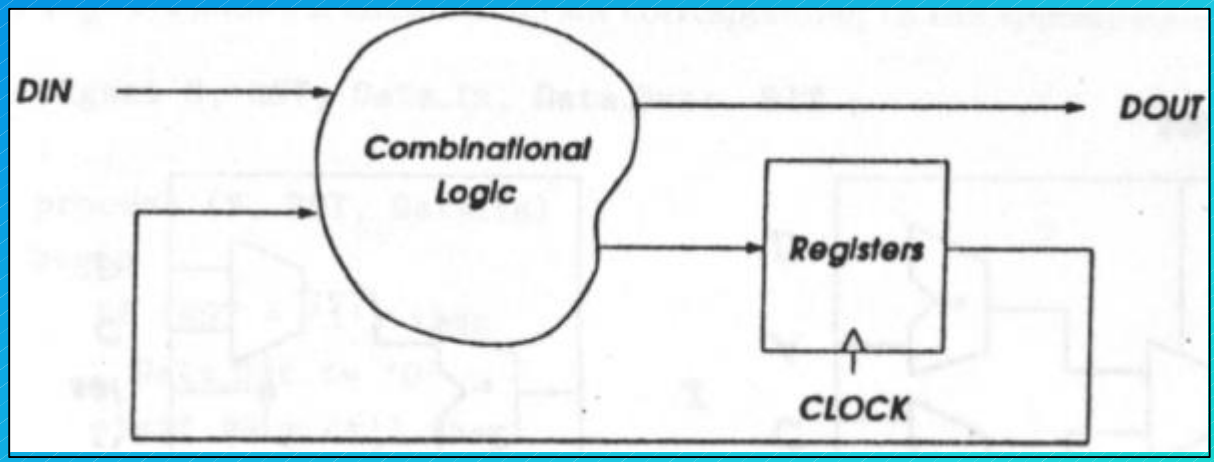
MODELLING A TEST BENCH

```
    U1 : FULL_ADDER port map ( A, B, Cin, Sum, Cout ) ;
    U2 : A <=  '0' ,
               '1' after 50 ns ,
               '0' after 100 ns ,
               '1' after 200 ns ;
    U3 : B <=  '0' ,
               '1' after 50 ns ,
               '0' after 110 ns ,
               '1' after 150 ns ;
    U4 : Cin <= '1' ,
               '0' after 10 ns ,
               '0' after 50 ns ,
               '1' after 200 ns ;
end testbench
```

You don't need to use identifiers (***U2, U3, U4***) for signal assignments.

MODELLING AT THE RT LEVEL

A ***Register Transfer Level*** (*RTL*) design consists of a set of registers connected by combinational logic as shown :



COMBINATIONAL LOGIC

A process without ***if signal leading edge*** (or ***falling edge***) statements or wait ***signal'event*** statements is called a ***combinational process***. All the sequential statements except ***wait*** statements, ***loop*** statements and ***if signal leading edge*** (or ***falling edge***) statements can be used to describe a combinational logic. Combinational logic does not have a memory to hold a value. Therefore a variable or signal must be assigned a value before being referenced. The following example describes a combinational logic:

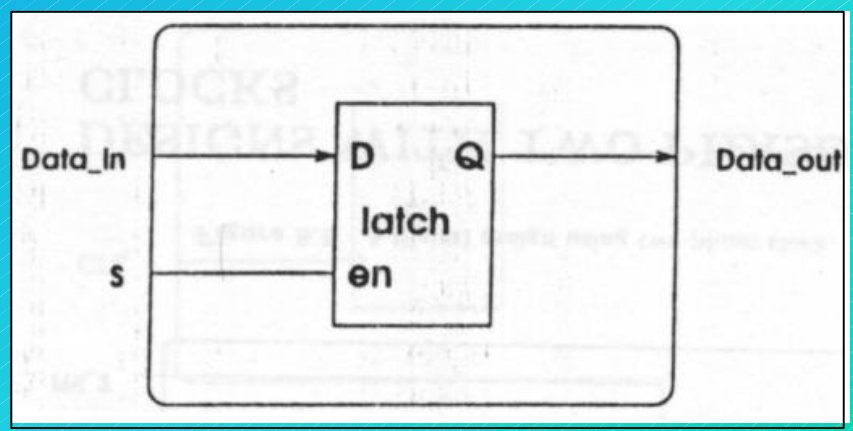
```
process (A, B, Cin)
begin
        Cout <= (A and B) or ((A or B) and Cin) ;
end process ;
```

All the input signals must be listed in the sensitivity list !!!

LATCHES

Flip-flops and ***latches*** are two commonly used one-bit memory devices. A flip-flop is an edge triggered memory device. A latch is a level sensitive memory device. In general latches are synthesized from incompletely specified conditional expressions in a combinational description. Any signal or variable that is not driven under all conditions becomes a latched element. Incompletely specified ***if*** and ***case*** statements create latches. For example the following if statement does not assign a value to DATA_OUT when S is not '1'. So synthesizer will create a latch.

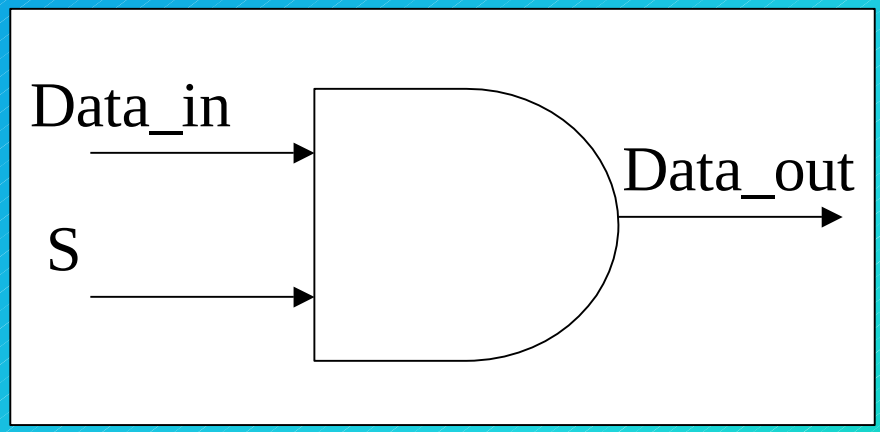
```
signal  S, Data_in, Data_out : bit ;
process (S, Data_in)
begin
      if (S = '1') then
            Data_out <= Data_in ;
      end if ;
end process ;
```



LATCHES

To avoid having a latch inferred, assign a value to all signal under all conditions. Adding an else statement to the previous example will cause the synthesizer to realize an ***and gate***.

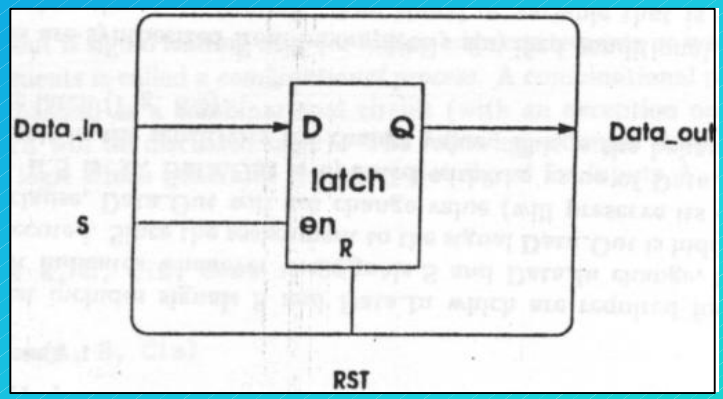
```
signal  S, Data_in, Data_out : bit ;
process (S, Data_in)
begin
      if (S = '1') then
            Data_out <= Data_in ;
      else
            Data_out <= '0' ;
      end if ;
end process ;
```



LATCHES

We can specify a latch with an asynchronous reset or an asynchronous preset.

```
signal  S, RST, Data_in, Data_out : bit ;
process (S, RST, Data_in)
begin
      if  (RST ='1') then
            Data_out <= '0' ;
      elsif (S = '1') then
            Data_out <= Data_in ;
      end if ;
end process ;
```

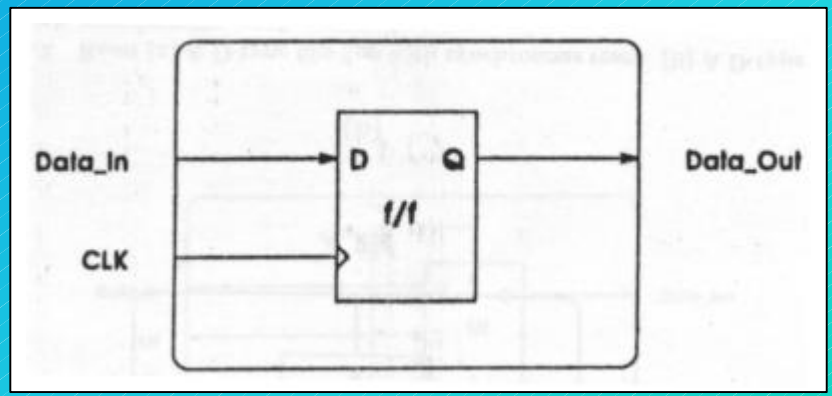


Instead of ***Data_out <= '0'*** assignment , we may assign ***'1'*** for asynchronous preset.

FLIP-FLOPS

A process with if signal leading edge (or falling edge) statements or wait signal'event statements is called a clocked process. An edge triggered flip-flop will be generated if a signal assignment is executed on the leading edge (or falling edge) of another signal.

```
signal  CLK, Data_in, Data_out : bit ;
process (CLK)
begin
      if (CLK'event and CLK = '1' ) then
            Data_out <= Data_in ;
      end if ;
end process ;
```



FLIP-FLOPS

Variables can also generate flip-flops. Since the variable is defined in the process itself, and its value never leaves the process, ***the only time a variable generates a flip-flop is when the variable is used before it is assigned in a clocked process !!!***

```
process (CLK)
variable a, b : bit ;
begin
        if (CLK'event and CLK = '1' ) then
                Data_out <= a ;
                a := b ;
                b := Data_in ;
        end if ;
end process ;
```

3 flip-flops are generated.

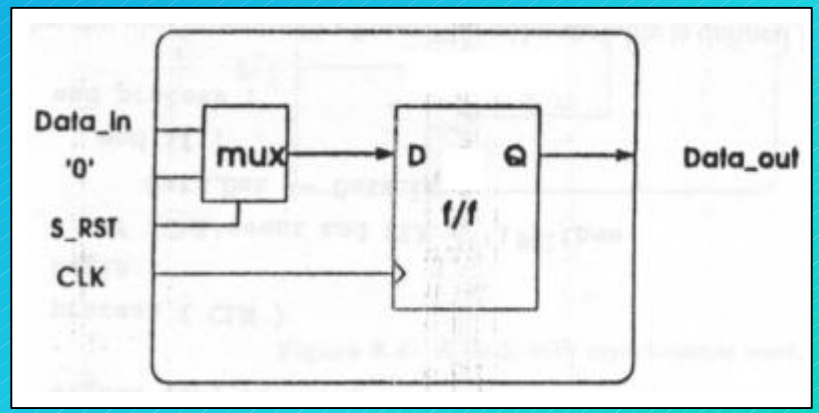
```
process (CLK)
variable a, b : bit ;
begin
        if (CLK'event and CLK = '1' ) then
                b <= Data_in ;
                a := b ;
                Data_out <= a ;
        end if ;
end process ;
```

1 flip-flop, 2 wires are generated.

SYNCHRONOUS SETS AND RESETS

Synchronous inputs set (preset) or reset (clear) the output of flip-flops while the clock edge is active. At all other times, changes on these inputs are not noticed by the memory element.

```
signal  CLK, S_RST, Data_in, Data_out : bit ;
process (CLK)
begin
      if  (CLK'event and CLK = '1') then
            if (S_RST = '1') then
                  Data_out <= '0' ;
            else
                  Data_out <= Data_in ;
            end if ;
      end if ;
end process ;
```

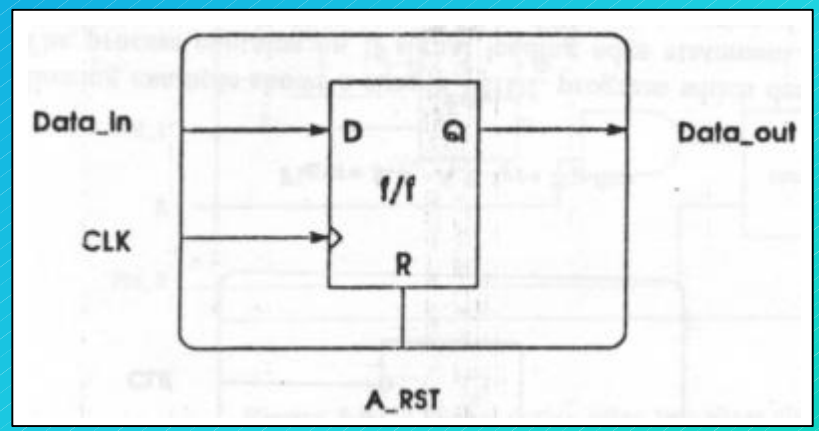


Note that it does not matter whether or not signals Data_in and S_RST are in the sensitivity list. Because their change does not result in any action in the first if statement.

ASYNCHRONOUS SETS AND RESETS

Asynchronous inputs set (preset) or reset (clear) the output of flip-flops independently of the clock.

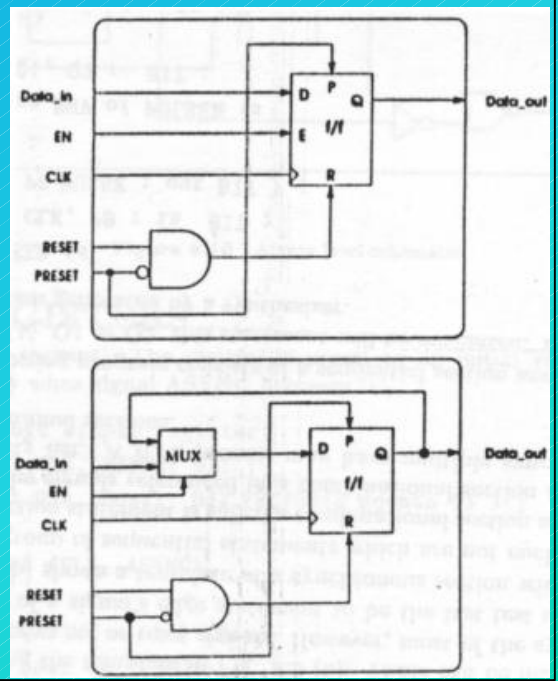
```
signal  CLK, A_RST, Data_in, Data_out : bit ;
process (CLK, A_RST)
begin
      if (A_RST = '0') then
            Data_out <= '0' ;
      elsif  (CLK'event and CLK = '1') then
            Data_out <= Data_in ;
      end if ;
end process ;
```



ASYNCHRONOUS SETS AND RESETS

It is possible to describe a flip-flop with more than one asynchronous inputs. In the following example we have 3 asynchronous inputs. Preset has precedence than reset signal. If there is not a flip-flop with an enable input in the library, then the generated circuit will be like the second one.

```
signal  CLK, RST, PRST, EN,  Data_in, Data_out : bit ;
process (CLK, RST, PRST, EN)
begin
        if (PRST = '1') then
                Data_out <= '1' ;
        elsif ( RST = '1') then
                Data_out <= '0' ;
        elsif  (CLK'event and CLK = '1') then
                if ( EN = '1' ) then
                        Data_out <= Data_in ;
                end if ;
        end if ;
end process ;
```



SYNCHRONOUS AND COMBINATIONAL RTL CIRCUITS

We can divide the statements of an RTL process into several ***synchronous*** and ***combinational sections***.

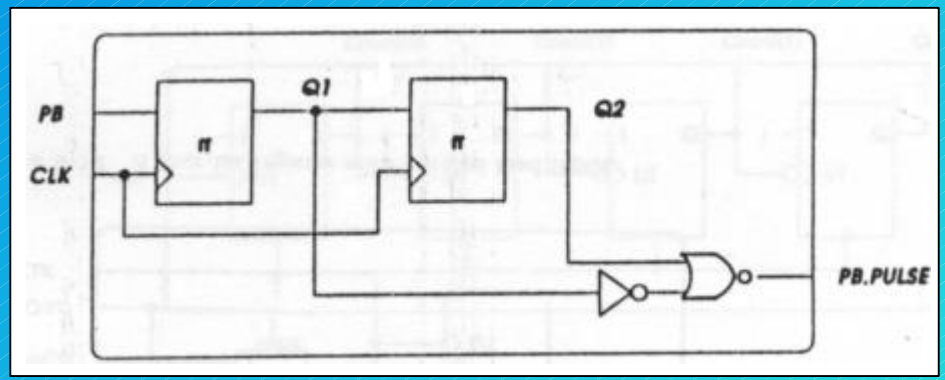
A synchronous section describes a sub-circuit whose behaviour will be evaluated only on the signal edges.

A combinational section describes a sub-circuit whose behaviour will be evaluated whenever there is a change on the signals of the sensitivity list.

All the signals referenced in a combinational section must be listed in the sensitivity list.

SYNCHRONOUS AND COMBINATIONAL RTL CIRCUITS

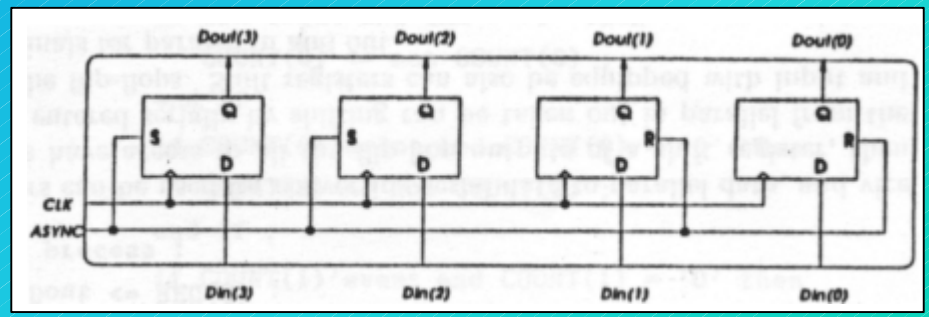
```
entity PULSER is
      port ( CLK , PB : in bit ;
            PB_PULSE : out bit ) ;
end PULSER ;
architecture BHV of PULSER is
      signal Q1, Q2 : bit ;
begin
      process ( CLK , Q1, Q2 )
      begin
            if (CLK'event and CLK = '1') then
                  Q1 <= PB ;
                  Q2 <= Q1 ;
            end if ;
            PB_PULSE <= (not Q1) nor Q2 ;
      end process ;
end BHV ;
```



REGISTERS

Various types of registers are used in a circuit. The following example shows a four-bit register which is asynchronously presets to “1100”.

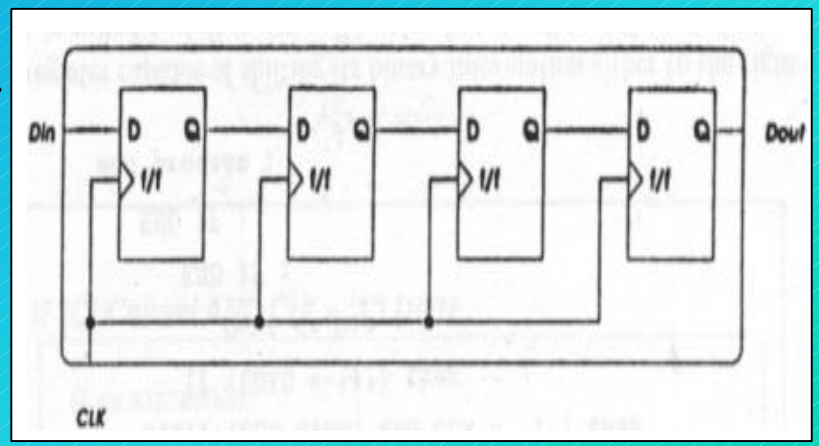
```
signal  CLK, ASYNC :  bit ;
signal Din, Dout : bit_vector (3 downto 0 );
process (CLK, ASYNC)
begin
      if (ASYNC = ‘1’) then
            Data_out <= “1100” ;
      elsif  (CLK’event and CLK = ‘1’) then
            Dout <= Din ;
      end if ;
end process ;
```



SHIFT REGISTERS

A register capable of shifting its binary information either to the right or to the left is called a ***shift register***. The logical configuration of a shift register consists of a chain of flip-flops connected in cascade, with the output of one flip-flop connected to the input of the next flip-flop. All flip-flops receive a common clock pulse that causes the data to shift from one stage to the next.

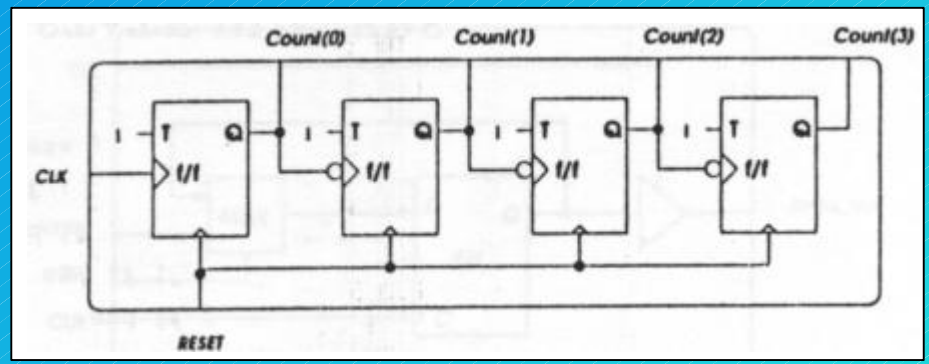
```
signal  CLK, Din, Dout :  bit ;
process (CLK)
      variable REG : bit_vector (3 downto 0) ;
begin
      if  (CLK'event and CLK = '1') then
            REG := Din & REG (3 downto 1) ;
      end if ;
      Dout <= REG (0) ;
end process ;
```



ASYNCHRONOUS COUNTERS

An asynchronous counter is the one whose state changes are not controlled by a synchronizing clock pulse.

```
signal  CLK, RESET :  bit ;
signal COUNT : bit_vector ( 3 downto 0) ;
process (CLK, COUNT, RESET)
begin
        if RESET = '1' then COUNT <= "0000" ;
        else
                if  (CLK'event and CLK = '1') then
                        COUNT (0) <= not COUNT (0) ;
                end if ;
                if  (COUNT(0)'event and COUNT(0) = '1') then
                        COUNT (1) <= not COUNT (1) ;
                end if ;
                if  (COUNT(1)'event and COUNT(1) = '1') then
                        COUNT (2) <= not COUNT (2) ;
                end if ;
                if  (COUNT(2)'event and COUNT(2) = '1') then
                        COUNT (3) <= not COUNT (3) ;
                end if ;
        end if ;
end process ;
```



SYNCHRONOUS COUNTERS

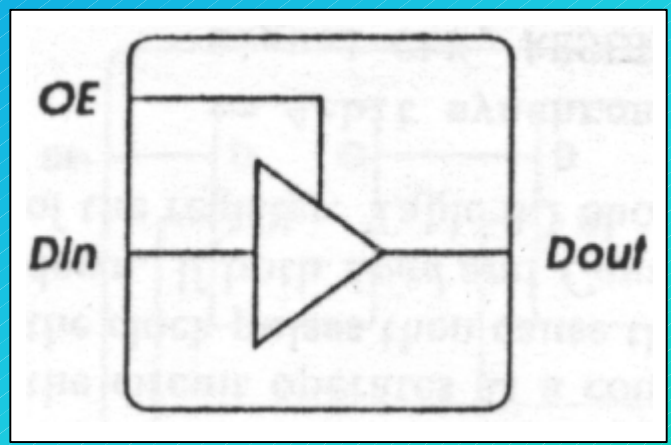
If all the flip-flops of a counter are controlled by a common clock signal, it is a synchronous counter..

```
signal  CLK, RESET , load , Count, UpDown :  bit ;
signal Datain : integer range 0 to 15 ;
signal Reg : integer range 0 to 15 := 0 ;
process (CLK, RESET)
begin
        if RESET = '1' then Reg <= 0 ;
        elsif   (CLK'event and CLK = '1') then
                if Load = '1' then
                        Reg <= Datain ;
                else
                        if Count = '1' then
                                if Updown = '1' then
                                        Reg <= (Reg +1) mod 16 ;
                                else
                                        Reg <= (Reg - 1) mod 16 ;
                                end if ;
                        end if ;
                end if ;
        end if ;
end process ;
```

TRI-STATE BUFFERS

Besides ***0*** and ***1***, there is a third signal value in digital systems : the ***high impedance*** state (*Z*). Among the predefined types of package STANDARD, there is no type to describe the high impedance value. ***STD_LOGIC type must be used !!!***

```
library IEEE ;
use IEEE . STD_LOGIC_1164.all ;
architecture IMP of TRI_STATE is
signal Din, Dout , OE: STD_LOGIC ;
begin
        process (OE, Din)
        begin
                if (OE = '0') then
                        Dout <= 'Z' ;
                else
                        Dout <= Din ;
                end if ;
        end process ;
end IMP ;
```



BUSSES

A ***bus*** system can be constructed with tri-state gates instead of multiplexers.

The designer must guarantee no more than one buffer will be in the active state at any given time. The connected buffers must be controlled so that only one tri-state buffer has access to the bus line while all other buffers are maintained in high impedance state.

Normally simultaneous assignment to a signal such as BusLine in the example is not allowed at the architectural level. However data types STD_LOGIC and STD_LOGIC_VACTOR can have multiple drivers.

BUSSES

```
library IEEE ;
use IEEE . STD_LOGIC_1164.all ;
entity BUS  is
        port ( S : in STD_LOGIC_VECTOR ( 1 downto 0) ;
                OE : buffer STD_LOGIC_VECTOR ( 3 downto 0) ;
                R0, R1, R2, R3 : in  STD_LOGIC_VECTOR ( 7 downto 0) ;
                BusLine : out STD_LOGIC_VECTOR ( 7 downto 0)) ;
end BUS ;
architecture IMP of BUS  is
begin
        process ( S )
        begin
                case ( S ) is
                        when “00” => OE <= “0001” ;
                        when “01” => OE <= “0010” ;
                        when “10” => OE <= “0100” ;
                        when “11” => OE <= “1000” ;
                        when others => null ;
                end case ;
        end process ;
```

BUSSES

```
process ( S )
begin
        case ( S ) is
                when "00" => OE <= "0001" ;
                when "01" => OE <= "0010" ;
                when "10" => OE <= "0100" ;
                when "11" => OE <= "1000" ;
                when others => null ;
        end case ;
end process ;
BusLine <= R0 when OE ( 0 ) = '1' else "ZZZZZZZZ" ;
BusLine <= R0 when OE ( 1 ) = '1' else "ZZZZZZZZ" ;
BusLine <= R0 when OE ( 2) = '1' else "ZZZZZZZZ" ;
BusLine <= R0 when OE ( 3 ) = '1' else "ZZZZZZZZ" ;
end IMP ;
```

