VHDL

As the size and complexity of digital systems increase, they cannot be designed manually; their design becomes highly complex. At their most detailed level, the circuit consists of millions of elements. So Computer Aided Design (CAD) tools are used in the design of such systems. One such a tool is a **Hardware Description Language (HDL).**

VHDL is an acronym for VHSIC Hardware Description Language (VHSIC- Very High Speed Integrated Circuit). It is a hardware description language that can be used to model a digital system.

BASIC LANGUAGE ELEMENTS

The basic elements of the language are – data objects, literals and operators.
Data objects-that store values of a given type.
Literals represent constant values.
Operatorsoperate on data object.

Identifiers
An identifier in VHDL is a sequence of one or more characters. There are two types if identifiers.

1. Basic identifiers and
2. Extended identifiers.

A basic identifier composed of sequence of legal character is an upper-case letter (A... Z), or a lower-case letter (a. .. z), or a digit (0 . . . 9) or the underscore (_) character.

Rules for naming an identifier:

1. The first character in an identifier must be a letter and the last character may not be an underscore.
2. Lower-case and upper-case letters are considered to be identical when used in an identifier;
3. Also,-two underscore characters cannot appear consecutively.

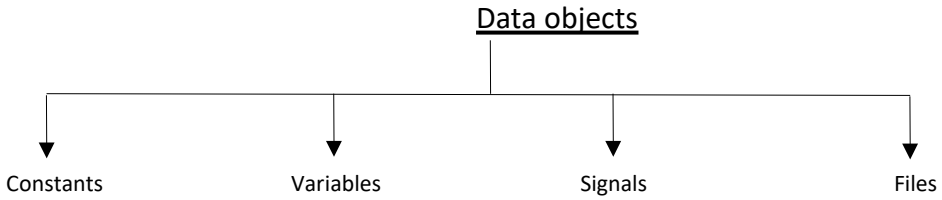
Example: Count, COUNT, and CouNT,
DRIVE_BUS, SelectSignalRAM_Address
SET_CK_HIGH CONST32_59 r2d2

An extended identifier is a sequence of characters written between two backslashes. Any of the allowable characters can be used. Within an extended identifier, lowercase and uppercase letters are considered to be distinct.

Example:
\TEST\, \-25\ , \2FOR$\ , \process\ etc.
Comments in a description must be preceded by two consecutive hyphens (--); the comment extends to the end of the line.
The language defines a set of reserved words. These words are called *keywords,* have a specific meaning in the language, and cannot be used as identifiers.

Data Objects

A data object holds a value of a specified type. It is created by means of an object declaration.



There are four classes of data objects, namely

1. Constants
2. Variables
3. Signals and
4. Files.

1. ***Constant:***An object of constant class can hold a single value of a given type. This value is assigned to the object before simulation starts and the value cannot be changed during the course of the simulation.

Syntax:

CONSTANT*const-name*: **TYPE** [:= value];

Examples:

constantRISE_TIME: TIME := 10ns;
constantBUS_WIDTH: INTEGER := 8:
constantNO_OF_INPUTS: INTEGER;

- The first declaration declares the object RISE_TIME that can hold a value of type TIME predefined and the value assigned to the object at the start of simulation is 10 ns.
- The second constant declaration declares constant BUS_WIDTH of type INTEGER with a value of 8.
- In the third declaration, the value of the constant has not been specified in this case. Such a constant is called a *deferred constant*

2. ***Variable:***An object of variable class can also hold a single value of a given type. But different values can be assigned to the object at different times using a variable assignment statement.

Syntax:

VARIABLE *var-name*: **TYPE** [:= initial value];

Examples:

variableCTRL_STATUS: BIT_VECTOR(10 **downto**0);
variableSUM: INTEGER **range Oto 100** := 10;

- The first declaration specifies a variable object CTRL_STATUS as an array of II elements, with each array element of type BIT.
- In the second declaration, an explicit initial value has been assigned to the variable SUM. When simulation starts, SUM will have an initial value of 10

3. ***Signal:***signals are data objects in VHDL that are used to model interconnections and capable of holding the list of values. An object belonging to the signal class has a past history of values, a current value, and a set of future values. Future values can be assigned to a signal object using a signal assignment statement. Signal objects can be regarded as wires in a circuit. Signal objects are typically used to model wires and flip-flops while variable and constant objects are typically used to model the behavior of the circuit.

Syntax:

SIGNAL *sig -name*: **TYPE** [:= initial value];

Examples:

signalCLOCK: BIT;
signalDATA_BUS: BIT_VECTOR(0 to 7);
signalGATE_DELAY: TIME := 10 ns;

The first signal declaration declares the signal object CLOCK of type BIT and gives it an initial value of '0' ('0' being the leftmost value of type BIT). The third signal declaration declares a signal object GATE_DELAY of type TIME that has an initial value of 10 ns.

4. **File:**An object belonging to file class contains a sequence of values. Values can be read or written Using file read procedures and write procedures.
   Syntax:

FILE *file-names* : file-**type-name**[[**open** mode] **is** string-expression];

Examples:
file STIMULUS:TEXT **open** READ_MODE **is** "/usr/home/jb/add.sti";
file VECTORS:BIT FILE **is** "/usr/home/james/add.vec";

In the first example, a file STIMULUS is declared to be a predefined file type TEXT; that is a file may contain indefinite number of strings. The mode value READ_MODE specifies that the file will be opened in read only mode and the string expression keyword **is** specifies the path name.

In the second example, VECTORS is declared as a file, containing an indefinite number of bit vectors. This declaration also specifies the link to the file in the host environment. Since no mode is specified, the default mode, READ_MODE is used.

Data Types

VHDL has a set of standard data types (predefined / built-in). It is also possible to have user defined data types and subtypes. i.e., in VHDL the data types can be classified into

1. Predefined type
2. User defined type

Some of the predefined data types in VHDL are: BIT, BOOLEAN and INTEGER.
The STD_LOGIC and STD_LOGIC_VECTOR data types are not built-in VHDL data types, but are defined in the standard logic 1164 package of the IEEE library.

- BIT: it is the simplest and most important data type for a digital system based on the logic values '0' and '1'. Any data object can be declared of the type bit before being used as Signal, Variable or Constant
- Boolean: It is predefined type that has values TRUE & FALSE. Some of the predefined operators are and,or,not,nand,nor.
- INTEGER: it is a predefined type with the set of values being integers in the range from $-(2^{31} - 1)$ to $+ (2^{31} - 1)$.

Data types (user defined)

ENUMERATION | INTEGER | PHYSICAL | FLOATING POINT | RECORD | ARRAY

1. Scalar types
2. Composite type
3. Access type
4. File type

1. Scalar Types

The values belonging to this type appear in sequential order. There are four different kinds of scalar types. These types are

a. Enumeration
b. Integer
c. Physical
d. Floating point.

Integer types, floating point types, and physical types are classified as *numeric types* since the values associated with these types are numeric.

a. Enumeration Types

An enumeration type declaration defines a type that has a set of user-defined values consisting of identifiers and character literals. i. e., it can take value from a defined list. These values may be integers, identifiers or single character literals.

SYNTAX:

TYPE name**IS**

Examples :

Type MVL **is** ('U','0','1','Z);
Type BIT**is** ('0','1');
TypeCAR_STATE **is** (STOP, SLOW, MEDIUM, FAST);
Type Boolean **is** (True ,False);
Type SEVERITY_LEVEL **is** (NOTE, WARNING, ERROR, FAILURE);
Type FILE_ OPEN_KIND **is** (READ_MODE, WRITE MODE , APPEND MODE);
Type FILE_ OPEN_KIND **is** (READ_MODE, WRITE MODE , APPEND MODE);

Summary of predefined enumerated data types

Type	Range of values
Character	ASCII characters, characters must be placed between single quotes.
Bit	'0' , '1'
Bit_vector	An array with each element of type bit
Boolean	FALSE, TRUE
File_open_kind	Read_mode, write_mode, append_mode
File_open_status	Open_ok, status _error, mode_error
Severity level	Note, Warning, Error, and Failure

b. Integer Types

An integer type defines all integer values whose set of values fall within the specified range. The values can be positive or negative. The range of the INTEGER type is in the range -(2^ 31 - 1) to +(2^31 - 1).

Values belonging to an integer type are called *integer literals.* Examples of integer literals are

56349 6E2 0 98_71_28

Literal 6E2 refers to the decimal value 6 * (10^2) = 600. The underscore (_) character can be used freely in writing integer literals and has no impact on the value of the literal; 98_71_28 is same as 987128.

c. Floating Point (Real) Types:

A floating point type has a set of values in a given range of real numbers.

SYNTAX:

TYPEreal_range**IS**range

Example:

typeTTL_VOLTAGE **is range** -5.5 **to** -1.4;
typeREAL_DATA **is range** 0.0 **to** 31.9;
variableLENGTH: REAL_DATA **range** 0.0 **to** 15.9;

Floating -point literals are values of a floating point type. Examples of floating point literals are

16.26 0.00.002 3_1.4_2

(Floating point literals differ from integer literals by the presence of the dot (.) character. Floating point literals can also be expressed in an exponential form. The exponent represents a power of ten and the exponent value must be an integer. Examples are

62.3 E-2 5.0 E+2

Integer and floating point literals can also be written in a base other than 10 (decimal). The base can be any value between 2 and 16. Such literals are called *based literals.* In this case, the exponent represents a power of the specified base. The syntax for a based literal is

base# based-value # -- form 1
base# based-value # **E** *exponent* -- form 2

d. Physical Types

A physical type contains values that represent measurement of some physical quantity, like time, length, voltage, and current. Values of this type are expressed as integer multiples of a base unit.

Example:

1. **type** CURRENT **is range 0 to** 1 E-9
   units
   nA; -- (base unit) nano-ampere
   uA = 1000 nA; -- micro-ampere
   mA = 1000 µA; --milli-ampere
   Amp = 1000 mA; -- ampere
   end units;
2. **type** TIME**is range 0 to -**1 E18 to 1 E18
   units
   pS; -- (base unit) Pico Sec
   nS = 1000pS;
   µS = 1000nS;
   mS = 1000µS;
   Sec= 1000mS
   Min= 60Sec
   end units;

CURRENT is defined to be a physical type that contains values from 0 nA to 10^9 nA. The base unit is a nano-ampere while all others are derived units.

2. Composite Types

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

- An array type and
- A record type.

2a. Array types:

Array is used to represent an object containing more than one values of similar type. For example, a register of eight bits can be represented as an object of the type array consisting of eight-bit values. So, eight values of type bit are grouped into a single object of the type array.

Example:

```
typeADDRESS_WORD is array (0 to 63) of BIT;
typeDATA_WORD is array (7 downto0) of MVL;
typeDECODE_MATRIX is array (POSITIVE range 15 downto1,
```

Address-Bus is one dimensional array object type that consists of element of type BIT.

Decode _Matrix is array of positive range 15 down to 1

2b. Record Types

Record type is used to group objects of different type together which can be referenced as a single group.

1. **type** MODULE **is**
   record
   SIZE: INTEGER **range** 20 **to** 200;
   CRITICAL_DLY: TIME;
   NO_INPUTS: PIN_TYPE:
   NO_OUTPUTS: PIN_TYPE;
   end record;

2. **type** DATE**is**
   record
   DAY: integer_range 1 to 31;
   MONTH:month_name;
   YEAR:integer_range0 to 4000;
   end record;

3. **type** FORECAST**is**
   record
   Temp: integer_range20 to 200;
   Day: Real;
   Condition: Boolean;
   end record;

Values can be assigned to a record type object using a single assignment statement.

3. Access Types

Access type points to address a particular type of object and is useful for accessing dynamically allocated objects.

Examples:

```
typePTR is access MODULE;
typeFIFO is array (0 to 63, 0 to 7) of BIT;
```

PTR is an access type whose values are addresses that point to objects of type MODULE.

4. File Types:

A file is a stream of values of specified type which can be read or written during simulation. Files belong to special kind of data type called “file type”.

Syntax

type*file-type-name* **Is file of** *type-name,*

The *type-name* is the type of values contained in the file.

Examples.

```
typeVECTORS is file of BIT_VECTOR;
typeNAMES 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.

OPERATORS

An operator is a logical or mathematical function which takes one or two values and produces a single result.
The predefined operators in the language are classified into the following five categories:

1. Logical operators
2. Relational operators
3. Shift operators.
4. Adding operators
5. Multiplying operators
6. Miscellaneous operators

The precedence of operators are from (6) to (1). That is, the miscellaneous operators have highest precedence while logical operators have lowest precedence. Operators in the same category have the same precedence and evaluation is done left to right. Parentheses may be used to override the left to right evaluation.

1. Logical Operators

The seven logical operators are

And or nand nor xorxnor not

These operators are defined for the predefined types BIT and BOOLEAN and one-dimensional arrays of BIT and BOOLEAN. During evaluation, bit values '0' and 1' are treated as FALSE and TRUE values of the BOOLEAN type, respectively. The result of a logical operation has the same type as its operands. The not operator is a unary logical operator and has the same precedence as that of miscellaneous operators.

2. Relational Operators

These are

= /= <<=>>=

The result types for all relational operations are always BOOLEAN (TRUE or FALSE). The = (equality) and the /= (inequality) operators are permitted on any type except file types. The remaining four relational operators are permitted on any scalar type.

For example,

BIT_VECTOR('0', '1', '1') < BIT_VECTOR('1', '0', '1')

is true, since the first element in the first array aggregate is less than the first element in the second aggregate.

3. Shift Operators: These are

sllsrlslasrarolror

Each of the operators takes an array of BIT or BOOLEAN as the left operand and an integer value as the right operand and performs the specified operation.

a) sll – (shift left logical):

It is a logical shift operation which shifts the bits to left and thus the vacated bits on the right are filled with zeros.
Example:
a<= 101001 ;b<= a sll 2 = 100100

b) srl – (shift right logical):

It is a logical shift operation which shifts the bits to right and thus the vacated bits on the left are filled with zeros.
Example:
a<= 101001; b<= a srl 2 = 001010

c) sla – (shift left arithmetic):

It is an arithmetic shift operation which shifts the bits to left and thus the vacated bits on the right are replicated with leftmost bit. (Filled value is right hand bit)

Example:

a<= 101001

b<= a sla2 = 100111

d) sra– (shift right arithmetic):

It is an arithmetic shift operation which shifts the bits to right and thus the vacated bits to left are replicated with leftmost bit. (Filled value is left hand bit)

Example:

a<= 101001 ;b<= a sra 2 = 111010

e) rol – rotate left:

It is a circular rotation operation which rotates the bits to left.

Example:

a<= 101001; b<= a rol 2 = 100110

f) ror – rotate right:

It is a circular rotation operation which rotates the bits to right.

Example:

a<= 101001; b<= a ror 2 = 011010

4. Adding Operators

These are

+ -&

The operands for the + (addition) and - (subtraction) operators must be of the same numeric type with the result being of the same numeric type. The operands for the & (concatenation) operator can be either a
I-dimensional array type or an element type.

Example,

'0' & '1' results in an array of characters "01".
'C' & 'A' &'T' results in the value "CAT".
"BA" & "LL" creates an array of characters "BALL".

5. Multiplying Operators

These are

*** / mod rem**

Multiplication and Division: The * (multiplication) and / (division) operators are binary operators in which both the operandsare of the same integer or floating point type. The result is also of the same type.

Remainder (rem): The rem (remainder) is a binary operator which returns the remainder of the division of two integer values. The result is also of the same type, but the sign of the first operand.

A **rem** B = A - (A / B) * B

Where:
A/B is an integer
(A rem B) has the same sign of A
Absolute value of (A rem B) < Absolute value of B

Example:

```
5 rem 3 = 2
(-5) rem 3 = -2
(-5) rem (-3) = -2
5 rem (-3) = 2
11 rem 4 = 3
(-11) rem 4 = -3
```

Modulus (Mod): The mod operator is a binary operator which works on integer values . The result of a mod operator has the sign of the second operand and is defined as

A **mod** B = A – B * N -For some integer N.

Where:
N is an integer
(A mod B) has the same sign of B
Absolute value of (A mod B) < Absolute value of B

Example:

```
5 mod 3 = 2
(-5) mod 3 = 1
(-5) mod (-3) = -2
5 mod (-3) = -1
9 mod 4 = 1
7 mod (-4) = -1
```

N is the smallest value for which the result is that of the second operand.

6. Miscellaneous Operators

The miscellaneous operators are

abs**

The **abs** (absolute) operator is defined for any numeric type.

The **** (exponentiation) operator is defined for the left operand to be of integer or floating point type and the exponent operator for the integer type only.

****

Operator priority:

Miscellaneous operators	** | **abs** | **not[Highest Precedence]**
Multiplying Operators	* | / | **mod** | **rem**
Adding Operators	+ | - | **&**
Shift Operators	**sll** | **srl** | **sla** | **sra** | **rol** | **ror**
Relational Operators	= | /= | < | <= | > | >=
Logical Operators	**and** | **or** | **nand**| **nor** | **xor** | **xnor** **[Lowest Precedence]**

BEHAVIORAL MODELING

In behavioral style of modeling the behavior of the entity is expressed using sequentially executed, procedural type code. The key features of this modeling are -

- The behavioral modeling describes the system by showing how the outputs behave according to the changes in the inputs.
- While describing in the behavioral modeling, it is not necessary to know the logic diagram of the system but it is required to know how the output behaves in response to the change in the input.
- In VHDL, **process** is the main behavioral description statement.
- The statements inside the process are sequential.

Sequential Vs concurrent statements:

Sequential statements are those statements, where the order or sequence of writing the statements is important and defines the step by step execution, followed one after the other.

In concurrent style of modeling the digital circuit, the order of statements is not important. Data flow and structural style modeling follow concurrent statements.

Every entity is represented using an entity declaration and at least one architecture body.

Entity Declaration:

An entity declaration describes the external interface of the entity, that is, it gives the black-box view. It specifies the name of the entity, the names of interface ports, their mode (i.e., direction), and the type of ports. The syntax for an entity declaration is

```
entity entity-name is
        [ generic ( list-of-generics-and-their-types ) ; ]
            [ port ( list-of-interface-port-names-and-their-types) ; ]
        [ entity-item-declarations ]
[ begin
        entity-statements]
      end [ entity-name ];
```

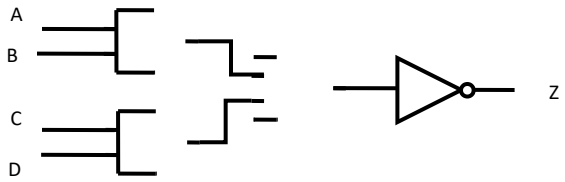
The *entity-name* is the name of the entity and the interface ports are the signals through which the entity passes information to and from its external environment. Each interface port can have one of the following modes:

1. **in:** Unidirectional port, indicating that the signal is an input and data can be written to.
2. **out:** Unidirectional port, indicating that the signal is an output of an entity whose value can be read. The value of an output port can only be updated within the entity model; it cannot be read.
3. **inout:** the value of a bidirectional port can be read and updated within the entity model.

4. **buffer:** the value of a buffer port can be read and updated within the entity model.

Example:

Consider an And-Or-Invert (AOI) circuit is shown in Fig. and its corresponding entity declaration is



entity AOI is

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

end AOI;

The entity declaration specifies that the name of the entity is AOI and that it has four input signals of type BIT and one output signal of type BIT.

Architecture Body

An architecture body describes the internal view of an entity. It describes the structure of the entity. Architecture consists of two portions:

- Architecture declaration and
- Architecture body.

The syntax of an architecture body is

architecture *architecture-name* **of** *entity-name* **is**

[*architecture-item-declarations]*

begin

Concurrent-statements; *these are —>*

Process-statement

Block-statement

Concurrent-procedure-call

Concurrent-assertion-statement

Concurrent-signal-assignment-statement

Component-instantiation-statement

generate-statement

end [*architecture-name*] ;

The architecture name is a user defined name of the architecture body. It can be same as entity name or different.

Process Statement

A process statement contains sequential statements that describe the functionality of a portion *of* an entity in sequential terms. The syntax of a process statement is

[*process-label:*] **process** [(*sensitivity-list)*]
[*process-item-declarations*]
begin
sequential-statements; *these are ->*
variable-assignment-statement
signal-assignment-statement
wait-statement
if-statement
case-statement
loop-statement
null-statement
exit-statement
next-statement
assertion-statement
procedure-call-statement
return-statement.
end process [*process-label]*;

Sensitivity list:

Sensitivity list is the set of signals to which the process is sensitive to (responsive). i.e., whenever an event occurs on any one of the signals in the sensitivity list, process comes into execution. The process is suspended only after executing all the statements inside the process in sequence.

For eg:

Consider the behavior model of AND gate. The signals A,B are included in the sensitivity list. So whenever the value of 'A' or 'B' or both changes from '0' to '1' or '1' to '0' , the process will start execution and the output of the gate is updated according to the expression C <= A and B;

Any change in the state of any element of the sensitivity list s treated as an event. The process is activated (initiated) on if an event occurs; otherwise process remains inactive. If the process has no sensitivity list, the process is executed continuously.

Variable Assignment Statement

Variables are the class of VHDL objects allowed only with the sequential style of modeling. Variables are objects that are used for the local storage within a process and subprogram alone. Inside a process local variables can be declared in the declarative part before the keyword **begin** to represent its local temporary values.

The first statement of the process statement is the variable assignment statement that assigns a value to variable **temp.** Variables can be declared and used inside a process statement. A variable is assigned a value using the variable assignment statement that typically has the form

***Variable-object : = expression*;**

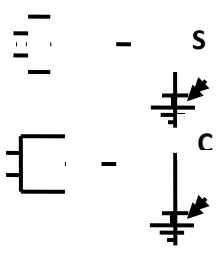
Differences between Signals and Variables

Signals	Variables
1. 1.These are VHDL objects used to represent wires and interconnections	These are temporary storage in VHDL
2. .Values of signals are updated only after default delta delay or specified user delay	Values of variables are updated immediately on the execution of variable assignment statement.
3. Require event scheduling and synchronizing of signal drivers.	No event scheduling and synchronizing is required.
4. Consume more memory space	Consume less memory space
5. Use of signals is allowed in styles of modeling	Variables are used only in behavioral modeling
6. Assignment operator is <=	Assignment operator is **:=**

[Write the VHDL code for the AOI circuit using Behavioral modeling].

```
entity AOI is
        port (A, B, C, D: in BIT; Z: out BIT);
end AOI;

architecture AOI of AOI is
        begin
            process (A, B, C, D)
            variable TEMPI ,TEMP2: BIT;
            begin
                TEMP1 := A and B;
                TEMP2:=C and D;
                TEMP1 := TEMP1 or TEMP2;
                    Z<= not TEMP1;
            end process;
            end AOI_SEQUENTIAL;
```

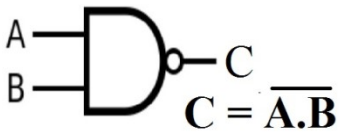
VHDL Behavioral description of Half adder



```
entity half adder is
port ( A : in bit;        B : in bit;
       Sum : out bit      Cout : out bit);
end  half_adder;

architecture adder of half_adder is
begin
      processs (A, B)
     begin
          sum <= A xor  B after 10ns;          -- signal assignment statement 1
          cout < = A and  B after 10ns          -- signal assignment statement 2
                                                -- with 10ns delay
    end process
end  adder;
```

Variable Assignment Statement: Examples

Write the VHDL code for two input nand gate using Behavioral modeling

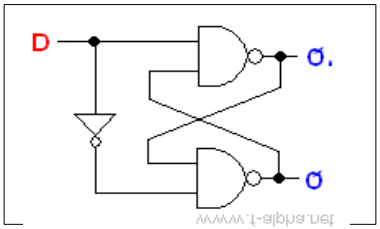



```
Entity nand2  is
Port      ( A : in bit;
            B: in  std-logic ;
            C : out std-logic);
End nand2;
```

```
Architecture nand2 of nand2 is
   Begin
        Process (A,B)

                Begin
                if A='1' and B='1' then
                C<= '0';
                 else
                C <= '1';
                End if;
                End process;
                End behavioral;
```

Write the VHDL code for D-latch using Behavioral modeling

```
entity D_latch is
                Port (D, Clk : in bit;
                      Q , Qbar : out bit);
end D_latch;
```



```
architecture behaviour of D_latch is
begin
                    process (D , Clk)
                    variable temp1 , temp2 : bit;
                    If Clk = '1' then
                    temp1 := D;                    --variable assignment statement.
                    temp2 := not temp1;            --variable assignment statement.
                          end if ;
                    Q <= temp1;                    -- value of temp1 is assigned to Q
                        Qbar <= temp2;             -- value of temp2 is assigned to Qbar.
end process;
end  D_latch;
```

Signal Assignment Statement:

A signal assignment statement can appear within a process or outside of a process. If it occurs outside of a process, it is considered to be a concurrent signal assignment statement. When a signal assignment statement appears within a process, it is considered to be a sequential signal assignment statement and is executed in sequence with respect to the other sequential statements that appear within that process.

When a signal assignment statement is executed, the value of the expression is computed and this value is scheduled to be assigned to the signal after the specified delay. If no delay is specified, the delay is assumed to be a default delta delay.

The syntax is

***Signal-object* <= *expression* [after *delay-value*];**

Example:

```
COUNTER <= COUNTER+ "0010";                    - Assign after a delta delay.
  PAR <= PAR xor DIN after 12 ns;
 Z <= (A0 and A1) or (B0and B1) or (C0 and C1) after 6 ns;
```

Write the VHDL code for D-latch using Behavioral modeling:

```
entity D_latch is
Port (D, En : in bit;
        Q : buffer bit;
      Qbar : out bit);
end D_latch;
architecture DL of D_latch is
begin
                    If En = '1' then
                      Q <= D;
                          Qbar <= not Q;
                    end if ;
end process;
end  DL;
```

Delta Delay

A *delta delay* *is* a very small delay. This delay models hardware where a minimal amount of time is needed for a change to occur. Delta delay allows for ordering of events that occur at the same simulation time during a simulation. Each unit of simulation time can be considered to be composed of an infinite number of delta delays. Therefore, an event always occurs at a real simulation time plus an integral multiple of delta delays.

For example, events can occur at 15 ns, 15 ns+IA, 15 ns+2A, 15 ns+3A,
22 ns, 22 ns+A, 27 ns, 27 ns+A, and so on.

Consider the AOI architecture

```
architecture AOI of AOI is
begin
        process (A, B, C, D)
        variable TEMPI ,TEMP2: BIT;
        begin
                TEMP1 := A and B;                -- statement 1
                TEMP2:= C and D;                 --statement 2
                TEMP1 := TEMP1 or TEMP2;         -- statement 3
                    Z<= not TEMP1;               --statement 4
        end process;
        end AOI_SEQUENTIAL;
```

Let us assume that an event occurs on input signal D (i.e., there is a change of value on signal D) at simulation time T. Statement I is executed first and TEMPI is assigned a value immediately since it is a variable. Statement 2 is executed next and TEMP2 is assigned a value immediately. Statement 3 is executed next which uses the values of TEMPI and TEMP2 computed in statements I and 2, respectively, to determine the new value for TEMPI. And finally, statement 4 is executed that causes signal Z to get the value of its right-hand-side expression after a delta delay, that is, signal Z gets its value only at time T+A; this is shown in Fig.

D

Z

Sequential statements:

The sequential statements exist inside the boundaries of a process statement as well as sub-programs. These are-

1. The variable assignment statement
2. the signal assignment statement
3. wait statement
4. if statement
5. Case statement
6. Null statement
7. loop statement
8. Exit statement
9. Next statement
10. Assertion statement
11. Report statement.

Wait Statement:

The wait statement is a statement that causes suspension of a process or a procedure.

WAIT statement exists in three forms as follows.

1. **wait on** signal_list;

   Eg: **wait on** S1, S2;

The process will be suspended on the wait statement and will be resumed when one of the S1 or S2 signals changes its value.

2. **wait until** condition;

Eg: **wait until** Enable = '1';

The wait statement will resume the process when the Enable signal changes its value to '1'.

3. **wait for** time;

Eg: **wait for** 50 ns;

A process containing this statement will be suspended for 50 ns.

4. **WAIT FOR 0:**

Syntax : Wait for 0ns

It means to wait for one delta cycle. This is a useful statement when the process is to be delayed.

For e.g.:

Wait 0: **process**
Begin
Wait on data
Sig_A <= data;
Wait for 0 ns;
Sig_B <= Sig_A;
End process;

The **Wait for 0ns** causes the process to suspend for 1Δ. SIG_A gets updated with its new value. Process resumes at 10 + 1Δ. SIG_B gets the new value of SIG_A at 10 + 2Δ. This is as shown below.

Data

SIG_A

SIG_B

If Statement:

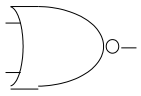
*The **if** statement is a statement that depending on the value of one or more corresponding conditions, selects for execution one or none of the enclosed sequences of statements,* IF statement exists in three forms.

1. **if** *boolean-expression then*
   sequential-statements
   end if;

Example1:

```
if SUM <= 100 then
SUM := SUM+10; end if;
```

Example2: Execution of nor gate:



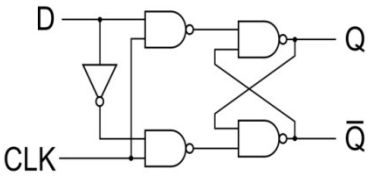
A	B	Y = A+B
0	0	1
0	1	0
1	0	0
1	1	0

```
entity NOR2 is
        port (A, B: in BIT; Z: out BIT);
 end NOR2;
architecture behaviour of NOR2 is
begin
        process (A, B)
                constant RISE_TIME: TIME := 10 ns;
                constant FALL_TIME: TIME := 5 ns:
                variable TEMP: BIT;
                        begin
                        TEMP := A nor B;
                        If  (TEMP = '1 ') then
                          Z <= TEMP after RISE_TIME;
                          else
                          Z <= TEMP after FALL_TIME;
                                end if;
                end process;
                end Behaviour;
```

2. **if** *boolean-expression then*
   sequential-statements
   elsif
   boolean-expression **then**
   sequential-statements
   else
   sequential-statements
   end if;

Example: Execution of D flip-flop:

```
Entity  dff is
PORT (d, clk, rst : in-std-logic;
        Q, qbar: out-std-logic );
End dff;
Architecture behavior of dff is
                Begin
                Process(rst, clk)
                        Begin
                        If rst = '0' then
                        Q = '0';
                        elsif clk 'event and clk ='1' then
                        Q = 'd';
                        End if;
End Process;
End behavior;
```



3 **if** *boolean-expression then*
sequential-statements
else
sequential-statement
end if;

Example: Execution of 4 bit up counter

```
Entity counter is
    Port(E,clk,rst : in_std_logic;
    Count: inout std_logic_vector (3 down to 0);
 End counter;

Architecture behavior of counter is
Begin
                Process(rst, clk)
        Begin
        If rst = '0' then
         Count <= "0000";
                elsif clk 'event and clk ='1' then
         Count <= count +1;
        Else
          Count <= count;
                End if;
End Process;
End behavior;
```

The **if** statement is executed by checking each condition sequentially until the first true condition is found; then, the set of sequential statements associated with this condition is executed. The **if** statement

can have zero or more **elsif c**lauses and an optional else clause. An **if** statement is also a sequential statement, and therefore, the previous syntax allows for arbitrary nesting of if statements.

(Refer more programs in 'Godse')

Case Statement:

The syntax is

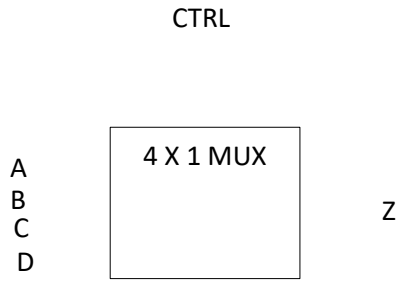
case *expression* **is**
when *choices* => *sequential-statements*
when *choices* => *sequential-statements*
[**when others** => *sequential-statements*]
.
.
..
end case;

The CASE statement executes the proper statement depending on the value of the input instruction. If the value of the instruction is one of the choices listed in the WHEN clauses then the statement following the when clause is executed.

If the value of the expression is outside the range of the choices given, then the expression following the OTHERS clause is executed.

Examples:

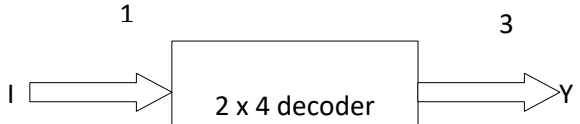
Write the VHDL code for MUX.




```
entity MUX is
        port (A, B, C, D: in BIT;
        CTRL: in BIT_VECTOR(0 to 1);
            Z: out BIT);
        end MUX;
        architecture BEHAVIOR of MUX is
        begin
    process (A, B, C, D, CTRL)
    begin
            case CTRL is
                when "00" => Z:= A:
                when "01" => Z := B;
                when "10" => Z := C;
                when "11" => Z := D;
                end case;
            end process
            end BEHAVIOR;
```

Ctrl lines		Inputs			
S_0	S_1	D	C	B	A
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Write the VHDL code for DECODER (2 x 4 decoder).



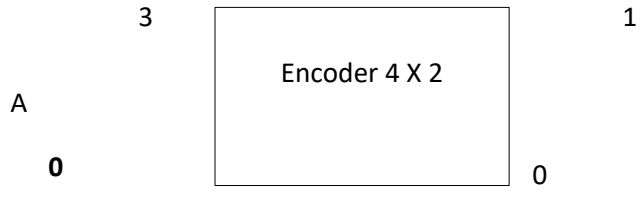
Entity decoder **is**

Port (I : in **std-logic-vector (1 downto 0)**;
y : out **std-logic-vector (3 downto 0));**
End decoder;

Architecture behavioral **of** decoder **is**
Begin
Process (I)
Case I **is**
When "00" => y := "0001",
When "01" => y := "0010",
When "10" => y := "0100",
When "11" => y := "1000",
Ende case;
End behavioral ;

Ctrl lines		Inputs			
I_1	I_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

VHDL code for ENCODER (4 x 2 encoder).



INPUTS				OUTPUTS	
A(3)	A(2)	A(1)	A(0)	B(1)	B(0)
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

entity encoder is
Port (A: in STD_LOGIC_ VECTOR (3 Down to 0);
B: out STD_LOGIC_VECTOR (1 Down to 0));
end encoder;

architecture Behavioral **of** encoder **is**
begin
process (A)
begin
case A is
When "0001" =>B <= "00";
When "0010" =>B <= "01";
When "0100" =>B <= "10";
When "1000" =>B <= "11";
When others =>B <= "00";
end case;
end process;
end Behavioral;

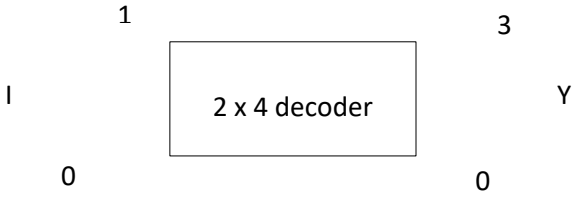
Null Statement

The statement **NULL** is a sequential statement that does not cause any action to take place and execution continues with the next statement.

It can be used to indicate that when some conditions are met, no action is to be performed. Such an application is useful in particular in conjunction with case statements to exclude some conditions.

Example:

Write the VHDL code for DECODER (2 x 4 decoder).




```
Entity  decoder is
Port  ( I : in std-logic-vector (1 downto 0);
        y : out  std-logic-vector (3 downto 0));
End decoder;

Architecture behavioral of decoder is
Begin
Process ( I)
                Case I is
                When "00" => y := "0001",
                When "01" => y := "0010",
                When "10" => y := "0100",
                When "11" => y := "1000",
                when others => null;
                End case;
End Process:
End behavioral ;
```

Ctrl lines		Inputs			
I_1	I_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Loop Statement:

Loop is a sequential statement. The LOOP statement is used whenever an operation needs to be repeated. Loop statements are used for iteration is needed in a model.

The syntax of a loop statement is

[*loop-label* :] *iteration-scheme* **loop**

sequential-statements

end loop [*loop-label*] ;

There are three types of iteration schemes.

- **for** *iteration scheme.*

  FOR identifier In range LOOP

  Statements;

  END LOOP;

- **while** *iteration scheme.*

  WHILE condition LOOP

  Statements;

END LOOP;

- The third form no iteration scheme is specified.

  LOOP

  Statements;

  END LOOP;

For Loop:

The **FOR-LOOP** statement is used whenever an operation needs to be repeated.
The **for** loop defines a loop parameter which takes on the type of the range specified.

for *identifier* **in** *range*

Example:

Write the VHDL code for factorial of a number using FOR LOOP:

```
Entity fact is
PORT (clk: in-std-logic;
        Factorial : out-integer );
End fact;

Architecture behavioral of fact is
            Begin
            Process (clk )
                  Begin
                  FACTORIAL := 1;
                  If clk ‘event and clk =’1’ then
                  for NUMBER in 2 to N loop
                        FACTORIAL := FACTORIAL * NUMBER;
                  End loop;
                  End IF;
                  End Process
End behavioral;
```

The body of the for loop is executed (N-1) times, with the loop identifier, NUMBER, being incremented by I at the end of each iteration.

While Loop:

WHILE loop differs from FOR loop as it repeats the sequential statements until a particular condition is met with. The syntax is

while *boolean-expression*

Example:

```
J:=0;SUM:=10;
while J < 20 loop
      SUM := SUM * 2;
```

```
        J:=J+3;
    end loop;
```

The statements within the body of the loop are executed sequentially and repeatedly as long as the loop condition, J < 20, is true. At this point, execution continues with the statement following the loop statement.

The third and final form of the iteration scheme is one where no iteration scheme is specified. In this form of **loop** statement, all statements in the loop body are repeatedly executed until some other action causes it to exit the loop. These actions can be caused by an **exit** statement, **a next** statement, or a return statement

Example:

```
SUM:=1;J:=0;
        L2: a label loop
        J:=J+21;
        SUM := SUM* 10;
        exit when SUM > 100;
end loop L2;
```

In this example, the exit statement causes the execution to jump out of loop L2 when SUM becomes greater than 100. If the exit statement were not present, the loop would execute indefinitely.

Exit Statement

The **EXIT** statement is a sequential statement that can be used only inside a loop. It is used to jump out of the loop conditionally or unconditionally and terminate the loop. The LOOP label in the EXIT statement identifies the particular loop to be exited

exit [*loop-label]* [**when** *condition]***:**

If no loop label is specified, the innermost loop is exited

Example:

```
SUM := 1; J := 0;
L3: loop
        J:=J+21;
        SUM := SUM* 10;
        if (SUM > 100) then
exit L3;
        end if;
end loop L3;
```

Next Statement

*The **next** statement is used to complete execution of one of the iterations of an enclosing loop statement. The completion is conditional if the statement includes a condition.*

Its syntax is

next [*loop-label]* [**when** *condition]*;

Example:

```
for J in 10 downto 5 loop
    if (SUM < TOTAL_SUM) then
    SUM := SUM +2;
    elsif (SUM = TOTAL_SUM) then
    next;
    else
    null;
    end if;
```

The difference between the Next statement and exit statement is that- the exit statement "exits" the loop entirely, while the next statement skips to the "next" loop iteration

Assertion Statement

Assertion statement checks whether a specified condition is true and reports an error if it is not.

The syntax is

assert *boolean-expression*

[**report** *string-expression*]

[**severity** *expression]*:

T**he** *assertion statement* has three optional fields and usually all three are used.

The condition specified in an ***assertion statement*** **must evaluate to a Boolean value** (true or false). If it is false, it is said that an assertion violation occurred.

The expression specified in the **report** clause **must be of predefined type STRING** and is a message to be reported when assertion violation occurred.

If the **severity** clause is present, it must **specify an expression of predefined type** SEVERITY_LEVEL, which determines the severity level of the assertion violation.

Example:

Functional errors, timing errors can be reported via **assert**:

```
entity RSFF is
    port( R,S, rst, CLK: in std_logic;
    Q,Qbar: out std_logic);
    End RSFF;
Architecture behavioral of RSFF is
    begin
        process (CLK , R,S)
    begin
        if (CLK‘ event and clock = ‘1’) then
```

```
            assert (S ='1' and R ='1');
            report "Undefined status "
            severity Error;
            end if;
            end process;
        end behavioural;
```

Report statement:

A report statement can be used to display a message. It is similar to an assertion statement but without the assertion check. The syntax is

report *string expression*
[**severity** *expression*];

When report statement is executed, it causes the specified string to be printed and the severity level to be reported to the simulator for appropriate action.

Examples:

1. **if** CLR = 'Z' **then**
   report *"signal CLR has a high impedance value";*
   end if;

2. **if** CLK /= '0' **and** CLK /= '1' **then**
   report *"CLK is neither '0' nor '1' ";*
   severity *ERROR;*
   end if;

More on Signal Assignment Statement:

Delay is the time gap between the giving the value at the input and the time at which the change due to input is reflected at the output. By default, the propagation delay of the circuit is present but VHDL gives the user the flexibility to specify the delay to manage the correct updation of values in case of concurrent statements which are all executed in parallel. There are two types of delay models in VHDL.

- Inertial and
- transport

Inertial Delay Model:

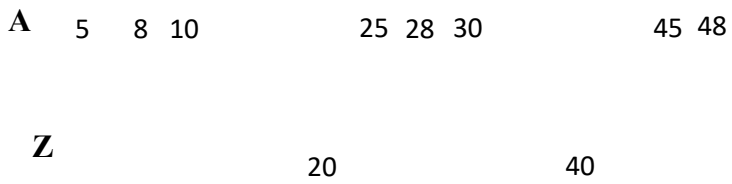
Inertial delay models the delays found in switching circuits. It represents the minimum length of time for which an input value must be stable change at the output. In addition, the value appears at the output after the specified delay. If the input is not stable for the specified time, no output change occurs.

The syntax is

Signal –object<= [[**reject** pulse-rejection-limit] **inertial**] expression **after** inertial-delay-value;

If no pulse rejection limit is specified, the default pulse rejection limit is the inertial delay value itself.

Example:
Consider a non-inverting buffer with an inertial delay of 10 ns.
Ie., Z = **reject** 4 ns **inertial** A **after** 10 ns

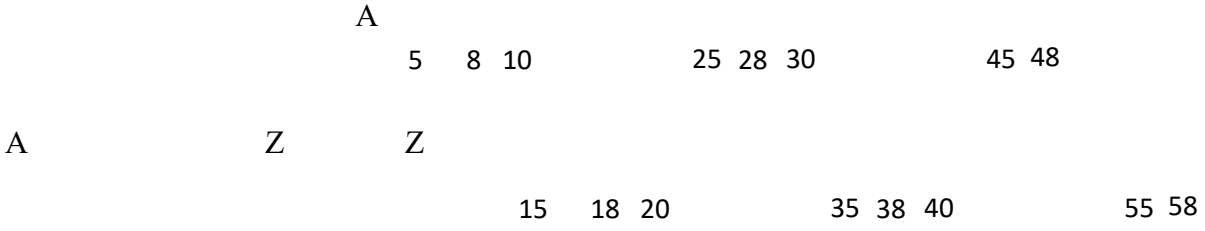


Events on signal A that occur at 5 ns and 8 ns are not stable for the inertial delay duration and hence do not propagate to the output. Event on A at 10ns remains stable for more than the inertial delay, and therefore, the value is propagated to the target signal Z after the inertial delay; Z gets the value 1' at 20 ns. Events on signal A at 25ns and 28 ns do not affect the output since they are not stable for the inertial delay duration. Transition 1' to '0' at time 30 ns on signal A remains stable for at least the inertial delay duration, and therefore, a '0' is propagated to signal Z with a delay of 10 ns; Z gets the new value at 40 ns. Other events on A do not affect the target signal Z.
Since inertial delay is most commonly found in digital circuits, it is the default delay model. This delay model is often used to filter out unwanted spikes and transients on signals.

Transport Delay Model
In Transport delay model the change in the input are transported to the output. The only delay that comes into play is the propagation delay and there is pulse rejection limit. The syntax is

Transport expression **after** inertial-delay-value;

Example:
Consider a non-inverting buffer with a transport delay of 10 ns.



In this case, spikes would be propagated through instead of being ignored as in the inertial delay case.

Differences between Inertial delay and Transport delay:

Inertial delay	Transport delay
1. This models propagation delay due to components as well as pulse rejection width.	This models delay due to wires or interconnections.
2. Any pulse whose duration is smaller than the propagation delay is not allowed to reach the output.	This delay allows every change to reach the output.
3. This most commonly used	Not very commonly used. .

Creating Signal Waveforms:

It is possible to assign multiple values to a signal, each with a different delay value.

For example,

PHASE1 <= '0', '1' **after** 8 ns, '0' **after** 13 ns, '1' **after** 50 ns;

When this signal assignment statement is executed, say at time T, it causes four values to be scheduled for signal PHASE l , the value '0' is scheduled to be assigned at time T+A, 1' at T+8 ns, '0' at T+13 ns, and 1' at T+50 ns. Thus, a waveform appears on the signal PHASE l as shown in Fig.

T T+8 T+13 T+50 ns

The syntax of signal assignment statement is

Signal –object<= [**transport**| **[reject** pulse-rejection-limit] **inertial**] waveform-element;

Each waveform element has a value part, specified by an expression (called the waveform expression in this text), and a delay part, specified by an after clause that specifies the delay. The delays in the waveform elements must appear in increasing order. A waveform element is of the form

expression **after** *time-expression*

Signal Drivers:

The effective value of a signal, if there is more than one assignment to the same signal within a process, can be obtained by creating the drivers.

Every signal assignment in a process creates a *driver* for that signal. The driver of a signal holds its current value and all its future values as a sequence of one or more transactions, where each transaction identifies the value to appear on the signal along with the time at which the value is to appear.

Example:

```
process
begin
. . .
        RESET <= 3 after 5 ns, 21 after 10 ns, 14 after 17 ns;
end process;
```

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

All transactions on the driver are ordered in increasing order of time

In the above example, when the signal assignment statement is executed, say at time T, three new transactions are added to the driver for the RESET signal. The first transaction is the current value of the signal.

When simulation time advances to T+5 ns, the first transaction is deleted from the driver and RESET gets the value of 3. When time advances to T+I0 ns, the second transaction is deleted and RESET gets the value of 21. When time advances to T+17 ns, the third transaction is deleted and RESET gets the value of 14.

Effect of Transport Delay on Signal Drivers

Consider an example of a process having three signal assignments to the same signal RX_DATA.

```
signal RX_DATA: NATURAL;
. . .
process
begin
        RX_DATA <= transport 11 after 10 ns:
        RX_DATA <= transport 20 after 22 ns;
        RX_DATA <= transport 35 after 18 ns;
end process;
```

Assume that the statements are executed at time *T.* The transactions on the driver for RX_DATA are created as follows.

When the first signal assignment is executed, the transaction, 11@T+10 ns, is added to the driver. After the second signal assignment is executed, the transaction, 20@T+ 22 ns, is appended to the driver since the delay of this transaction (= 22 ns) is larger than the delay of the pending transactions on the driver. The driver for RX_DATA looks like this

RX_DATA	*Curr @ now*	*11@T+ 10ns*	*20@T + 22ns*

When the third signal assignment statement is executed, the new transaction, 35@T+18 ns, causes the 20@T+22 ns transaction to be deleted and the new transaction is appended to the driver. Because the delay for the new transaction (=18 ns) is less than the delay of the last transaction sitting on the driver (=22 ns). This effect is caused because transport delay is used. In general, a new transaction will delete all transactions sitting on a driver that are to occur at or later than the delay of the new transaction. Therefore, the driver for RX_DATA is changed to

RX_DATA	*Curr @ now*	*11@T+ 10ns*	*35@T + 18ns*

Effect of Inertial Delay on Signal Drivers

When inertial delays are used, both the signal value being assigned and the delay value affect the deletion and addition of transactions. If the delay of the new transaction is earlier than an existing transaction, the latter is deleted and the new one is added at the end of the driver, regardless of the signal values of the two transactions

On the other hand, if the delay of the new transaction is greater than an already existing one, the signal values of the two transactions are compared. If they are the same, the new transaction is simply added at the end of the driver, if not, the existing one is deleted before adding the new transaction. Deletion occurs for every existing transaction with a signal value that is different from the new transaction.

Example:

Consider the following process statement.

```
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 transaction, 11@10 ns, first gets added to the driver. The second transaction, 22@20 ns, causes the 11@10 ns transaction on the driver to be deleted because the signal value, that is, 22, of the new transaction is different from the value of the transaction on the driver, that is, 11. The state of the driver at this point is

TX_DATA	*curr@now*	*22@ 20ns*

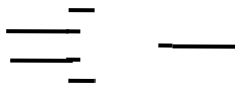
The execution of the third signal assignment causes the transaction 22@20 ns to be deleted from the driver, since the delay of the new transaction (=15 ns) is less than the delay of the transaction on the driver (similar to the transport delay case). The final status of the driver is

TX_DATA	*curr@now*	*35@15ns*

Dataflow Modeling

Dataflow modeling describes the architecture of the entity under design without describing its components in terms of flow of data from input towards output. This style is nearest to RTL description of the circuit. Dataflow modeling is concurrent style of modeling in VHDL, that is, unlike behavioral modeling the order of statements is not important

Example 1: Two input OR gate

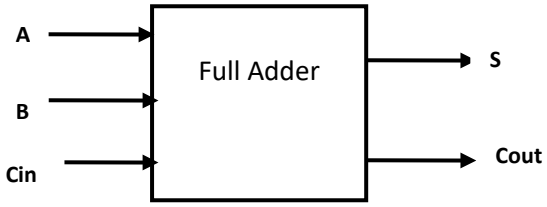


```
entity OR2 is
    port (signal A, B: in BIT; signal Z: out BIT);
end OR2;
        architecture data-flow of OR2 is
begin
Z <= A or B after 9 ns;
   End OR2;
```

In this example, the architecture body contains a single concurrent signal assignment statement. The interpretation of this statement is that, whenever an event (change of value) occurs on either A or B, the expression on the right is evaluated and the value is scheduled to appear on signal Z after a delay of 9 ns.

Example 2: 1 bit Full adder

.




```
entity FULL_ADDER is
port (A, B, CIN: in BIT; SUM, COUT: out BIT);
end FULL_ADDER;
        architecture data-flow of FULL_ADDER is
begin
        SUM <= (A xor B) xor CIN after 15 ns;
        COUT <= (A and B) or (B and CIN) or (CIN and A)
                                after 10 ns;
end  FULL_ADDER
```

Two signal assignment statements are used to represent the dataflow of the FULL_ADDER entity. Whenever an event occurs on signals A, B, or CIN, expressions of both the statements are evaluated and the value to SUM is scheduled to appear after 15 ns while the value to COUT is scheduled to appear after 10 ns. The after clause models the delay of the logic represented by the expression.

Concurrent versus Sequential Signal Assignment:

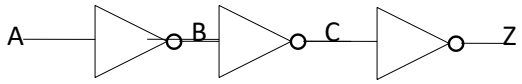
Concurrent signal assignment	Sequential signal assignment
Signal assignment statements that appear outside of a process are called *concurrent* signal assignment statements.	Signal assignment statements that appear within the body of a process statement are called *sequential* signal assignment statements,
Concurrent signal assignment statements are event triggered, that is, they are executed whenever there is an event on a signal that appears in its expression Example: Architecture df of csa is Begin A<= b; Z<= A; End df; When an event occurs on signal B at time T, signal A gets the value of B after delta delay of Δ. The	Sequential signal assignment statements are not event triggered and are executed in sequence in relation to the other sequential statements that appear within the process. Example: Architecture behavioral of ssa is Begin Process (B) Begin A<= b; Z<= A;

simulation time advances and the signal A will get the new value at T + Δ. This new value of A will trigger the second signal which will cause the new value of A to be assigned to Z at the time T +2Δ.	End process; End behavioral ; Whenever signal B has an event, the first signal assignment statement is executed and then the second signal assignment statement is executed, both in zero time. However, signal A is scheduled to get its new value of B only at time T+Δ and Z is scheduled to be assigned the old value of A (not the value of B) at time T+Δ also.

Delta Delay Revisited

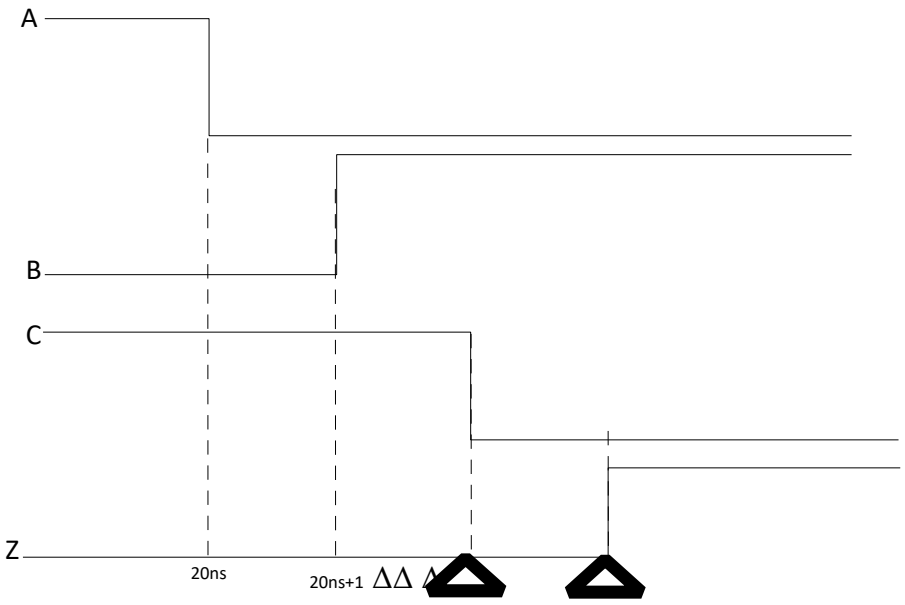
In a signal assignment statement, if no delay is specified or a delay of 0ns is specified, a delta delay is assumed.

Example 1:

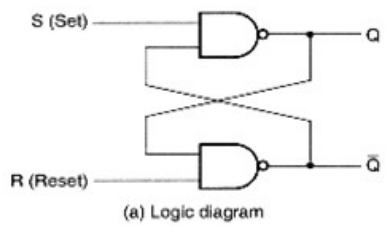



```
        entity FAST_INVERTER is
        port (A: in BIT; Z: out BIT);
        end FAST_INVERTER;
architecture DELTA_DELAY of FAST_INVERTER is
        signal B, C: BIT;
begin
        Z <= not C; - signal assignment #1
        C <= not B; - signal assignment #2
        B <= not A; - signal assignment #3
end DELTA_DELAY;
```

When an event occurs on signal A, say at 20 ns, the third signal assignment is triggered which causes signal B to get the inverted value of A at 20ns+1Δ. When time advances to 20ns+1Δ, signal B changes. This triggers the second signal assignment, causing signal C to get the inverted value of B at 20ns+2Δ. When simulation time advances to 20ns+2Δ, the first signal assignment is triggered causing Z to get a new value at time 20 ns+3Δ. Even though the real simulation time stayed at 20 ns, Z was updated with the correct value through a sequence of delta-delayed events. This sequence of waveforms is shown below



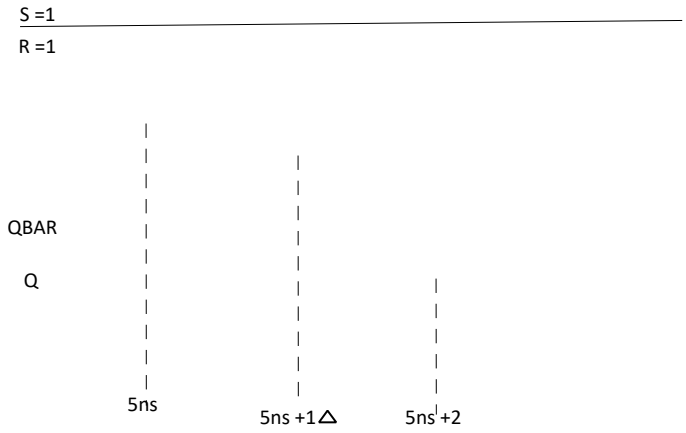
Example 2: RS latch



(a) Logic diagram

```
entity RS_LATCH is
port (R, S; in BIT := '1'; Q: buffer BIT := '1';
          QBAR: buffer BIT: = '0');
end RS_LATCH;
        architecture DELTA of RS_LATCH is
begin
        QBAR <= R nand Q;
        Q <= S nand QBAR;
end DELTA;
```

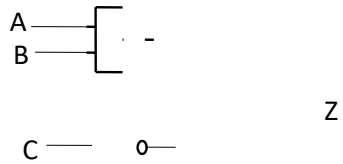
At start of simulation, both R and S have value' When the signal R changes from '1' to '0' at 5 ns. The following diagrams illustrate the event that occurs as a result of change in the value of R. This shows that the output gets stabilized after two delta delays. The circuit stabilizes with the final values of Q & Qbar being '0' and '1' respectively.



Multiple Drivers

Each concurrent signal assignment statement creates a driver for the signal being assigned. If the signal has more than one driver then that signal is said to have multiple drivers.

For e.g.:




```
entity md  is
        port (A, B, C: in BIT; Z: out BIT);
end md;
architecture df of md is
begin
        Z <= A and B after 10 ns;
        Z <= not C after 5 ns;
        End df;
```

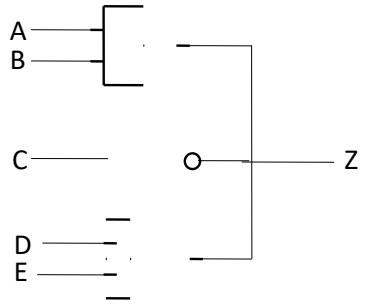
The effective value of Z is determined by using the user defined **"Resolution Function"**. This considers the value of both the drivers for Z and determines the effective value. A signal with more than one driver must have a resolution function associated with it, otherwise, it is an error. Such a signal is called a *resolved* signal

A resolution function consists of a function that is called whenever one of the drivers for the signals has an event occurring on it. When the resolution function is executed it returns a single value from all the values of the drivers. This is the new value of the signal.

Example: wired-OR, wired-AND, average value of a signal and so

Example:

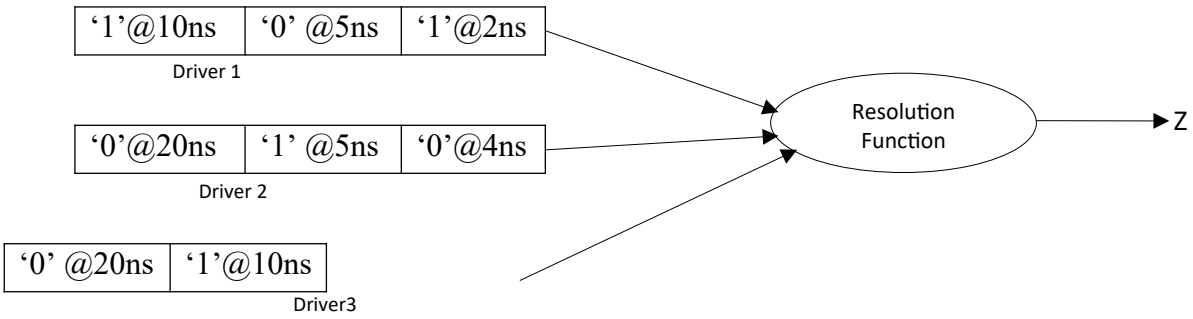
Consider the following circuit.




```
entity md  is
port (A, B, C,D,E: in BIT;
      Z: out BIT);
end md;
        architecture df of md is
        begin
        Z <= A and B after 10 ns;
        Z <= not C after 5 ns;
        Z <= A or B after 15 ns;
        End df;

Z <= '1' after 2 ns, '0' after 5 ns, '1' after 10 ns;
Z <= '0' after 4 ns, '1' after 5 ns, '0' after 20 ns;
Z <= '1' after 10 ns, '0' after 20 ns;
```

In this case, there are three drivers for signal Z. Each driver has a sequence of transactions where each transaction defines the value to appear on the signal and the time at which it is to appear. The resolution function resolves the value for the signal Z from the current value of each of its drivers. This is shown pictorially.

The value of each driver is an input to the resolution function and based on the computation performed within the resolution function, the value returned by this function becomes the resolved value for the signal.



The resolution function is associated with the signal by specifying the name in the signal declaration. (signal declaration could wired-or, wired and, average value etc.,) considering the wired-OR operation for the above example, the correct way if representing it is

```
 entity md  is
        port (A, B, C,D,E: in BIT;
                        Z: out  wired-OR BIT);
end md;
architecture df of md is
begin
        Z <= A and B after 10 ns;
        Z <= not C after 5 ns;
        Z <= A or B after 15 ns;
Z <= '1' after 2 ns, '0' after 5 ns, '1' after 10 ns;
Z <= '0' after 4 ns, '1' after 5 ns, '0' after 20 ns;
Z <= '1' after 10 ns, '0' after 20 ns;
```

End df;

In the example of architecture **md**, the resolution function is invoked at time 2 ns with driver values '1', '0', and '0' (drivers 2 and 3 have '0' because that is assumed to be the initial value of Z). The function, WIRED_OR, is performed and the resulting resolved value of '1' is assigned to Z at 2 ns. Signal Z is scheduled to have another event at 4 ns, at which time the driver values, '1', '0', and '0', are passed to the resolution function which returns the value of '1' for signal Z. At time 5 ns, the driver values, '0', '1', and '0' are passed to the resolution function which returns the value '1'. At 10ns, the driver values, '1', '1', and '1' are passed to the resolution function. Finally at time 20 ns, the driver values, '1', '0', and '0' are passed to the resolution function to determine the effective value for signal Z, which is 1'.

Delay (ns)	Driver 1	Driver 2	Driver 3	Z (Wired-AND)	Z (Wired-OR)
2	1	0	0	0	1
4	0	0	0	0	0
5	0	1	0	0	1
10	1	0	1	0	1
20	0	0	0	0	0

There are two forms of the concurrent signal assignment statement: the conditional signal assignment statement and the selected signal assignment statement.

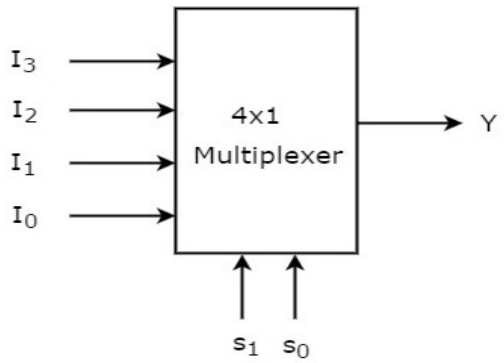
Conditional Signal Assignment Statement

The conditional signal assignment statement selects different values for the target signal based on the specified, different, conditions. A typical syntax for this statement is

Target - signal <= [*waveform-elements* **when** *condition* **else**]
[*waveform-elements* **when** *condition* **else]**
. . .
waveform-elements;

That is, whenever an event occurs on a signal used either in any of the waveform expressions or in any of the conditions, the conditional signal assignment statement is executed by evaluating the conditions one at a time.

Example: Consider 4 x 1 mux.



Ctrl lines		Inputs			
S_0	S_1	I3	I2	I1	I0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

DATAFLOW MODELING

```
entity MUX  is
port (I0,I1,I2,I3,S0,S1: in BIT;
    Y: out  BIT);
end md;

 architecture df of MUX is
begin

Y <= I0 after 10ns when S0 = '0' and S1 = '0' else
      I1 after 10ns when S0 = '1' and S1 = '0' else
      I2 after 10ns when S0 = '0' and S1 = '1' else
      I3 after 10 ns;
```

NOTE: Refer manual for selected signal assignment for MUX

BEHAVIORAL MODELING

```
entity MUX  is
port (I: in BIT_vector (3 down to 0);
          S: in BIT_vector (1 down to 0)
          Y: out  BIT);
end MUX;
 architecture behavioral of MUX is
begin
process
begin
        if S0 = '0' and S1 = '0' then
        Y<= I0 after 10 ns;
        Elsif  S0='1'and S1='0' then
        Y<= I1 after 10ns;
        Elsif S0='0' and S1 = '1' then
        Y<= I2 after 10 ns;
        else Y<= I3 after 10 ns;
end if;
wait on I0, I1, I2, I3, S0, S1;
end process;
end behavioral;
```

In the example of data flow modeling, the statement is executed any time an event occurs on signals I0, I1, I2, I3, S0, or S1. The first condition (S0='0' and S1='0') is checked; if false, the second condition (S0='1' and S1='0') is checked; if false, the third condition is checked; and so on. Assuming S0='0' and S1='1', then the value of IN2 is scheduled to be assigned to signal Y after 10 ns.

Selected Signal Assignment Statement:

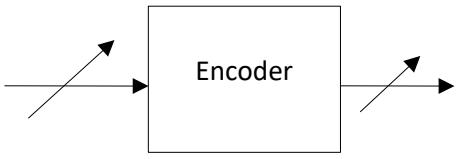
The selected signal assignment statement selects different values for a target signal based on the value of a select expression (it is like a case statement). The syntax for this statement is

with *expression* **select** — [This is the select expression].
target-signal <= *waveform-elements* **when** *choices,*
waveform-elements **when** *choices,*
...
waveform-elements **when** *choices ;*

Whenever an event occurs on a signal in the select expression or on any signal used in any of the waveform expressions, the statement is executed. The choices are not evaluated in sequence. All possible values of the select expression must be covered by the choices that are specified not more than once. Values not covered explicitly may be covered by an "others" choice.

Example:

Consider a 4 x 2 Encoder:



Inputs				Outputs	
I3	**I2**	**I1**	**I0**	**Y1**	**Y0**
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

```
entity encoder   is
port (I: in std_logic_vector(3 down to 0);
      Z: out std_logic_vector(1down to 0);
End encoder;
Architecture df of encoder is
Begin
With I select
Y<= "00" when "0001",
    "01" when "0010",
    "10" when "0100",
    "11" when "1000",
    "00" when others;
End df;
```

Conditional signal assignment

```
Y <= "00" when  I  "0001" else ,
     "01" when  I  "0010" else,
     "10" when  I  "0100" else,
     "11" when  I  "1000" else,
     "00" when  others;
```

Block Statement:

A block statement is a concurrent statement. There are two types of block statement –

1. Simple block
2. Guarded block

Simple BLOCK

The BLOCK statement, in its simple form, represents only a way of locally partitioning the code. It allows a set of concurrent statements to be clustered into a BLOCK, with the purpose of turning the overall code more readable and more Manageable.

Its syntax is shown below.

```
label: BLOCK
        [Declarative part]
BEGIN
        (concurrent statements)
END BLOCK label;
```

Guarded BLOCK

A guarded BLOCK is a special kind of BLOCK, which includes an additional expression, called guard expression. A guarded statement in a guarded BLOCK is executed only when the guard expression is TRUE.

```
Guarded BLOCK:
        label: BLOCK (guard expression)
                [declarative part]
        BEGIN
                (concurrent guarded and unguarded statements)
        END BLOCK label;
```

Example: D- flip-flop. with a Guarded BLOCK

In it, clk='1' is the guard expression, while q<=GUARDED d is a guarded statement.
Therefore, q<=d will only occur if clk='1'.

```
entity D_FLIP_FLOP is
        port (D, CLK: in BIT; Q, QBAR: out BIT);
end D_FLIP_PLOP;
architecture DFF of D_FLIP_FLOP is
begin
```

```
        b1: BLOCK (clk='1')
begin
        q <= GUARDED D;
        end block b1;
end DFF;
```

Concurrent Assertion Statement:

A concurrent assertion statement has exactly the same syntax as a sequential assertion statement assertion statement. The semantics of a concurrent assertion statement are as follows. Whenever an event occurs on a signal in the Boolean expression of the assertion statement, the statement is executed.

Example: Consider SR Flip-flop.

```
entity SR is
        port (S, R: in BIT; Q, NOTQ: out BIT);
end SR;
architecture df of SR is
begin
        assert (not(S = '0' and R = '0'))
        report "Not valid inputs: R and S are both low"
        severity ERROR;
        end df;
```

Anytime an event occurs on either of the signals, S or R, the assertion statement is executed and the Boolean expression checked. If false, the report message is printed and the severity level is reported to the simulator for appropriate action.

The unaffected value:

It is possible to assign a value of unaffected to a signal in a concurrent signal assignment statement. Such an assignment causes no change to the driver for the target signal.

Example: Consider 4 x 2 Encoder.

```
 entity encoder   is
port (I: in std_logic_vector(3 down to 0);
      Z: out std_logic_vector(1down to 0);
End encoder;

Architecture beh of encoder is
Begin
With I select

Y<= “00” when “0001”,
     “01” when “0010”,
     “10” when “0100”,
      “11” when “1000”,
      “00” when others;    (unaffected when others.)
End beh;
```

Value of a signal

A signal gets its value from its drivers. Every concurrent signal assignment statements create a driver for the target signal. All signal assignments in a process that assign to a particular signal create one driver for the signals.

In a given VHDL description, if a signal has more than one driver, the resolution function is necessary. This function is associated with the current value of all drivers for a signal and the return value of the function becomes the effective value of the signal.

Structural Modeling

Structural modeling is the simplest style of modeling. It describes the circuit design in terms of components. In this style of modeling, the top-level entity is partitioned into smaller lower level entities, each of which is known as the component.

Every component in VHDL is specified with its component declaration and mapped to top-level entity using component instantiation.

Component Declaration

Any entity before using the components is required to declare the components in the architecture – declarative section. The syntax of a simple form of component declaration is

```
component component-name
        port ( list-of-interface-ports ) ;
end component;
```

Example 1:

```
component NAND2
        port (A, B: in MVL; Z: out MVL);
end component;
```

Example 2:

```
component MP
        port (CK, RESET, RD, WR: in BIT;
                DATA_BUS: inout INTEGER range 0 to 255;
                ADDR_BUS: in BIT_VECTOR(15 downto 0));
end component;
```

Example 3:

```
component AND2
port (X, Y: in BIT: Z: out BIT):
end component;
```

Example 4:

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

Example 5

```
component NOR2
port (A, B: in BIT; Z: out BIT);
end component;
```

Component Instantiation:

Component instantiation is creating an instance of the specific component by associating its local ports to signals of the top-level entity. It puts the components used in the executable form for the top-level

entity. PORT MAP statement is used for component instantiation. A format of a component instantiation statement is

Component-label: component-name port **map** (association-*list) ',*

- The *component-label* can be any legal identifier
- The *component-name* must be the name of a component declared earlier using a component declaration.
- The *association-list* associates signals in the entity, called ***actuals,*** with the ports of a component, called ***formals*** .
- An actual may be a signal. An actual for an input port may also be an expression. Actual may also be the keyword OPEN to indicate a port that is not connected.

There are two ways to perform the association.

1. Positional association,

2. Named association.

In positional association, an *association-list is* of the form

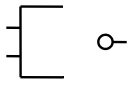
actual1, actual2, actual3. . . actual

Each actual in the component instantiation is mapped by position with each port in the component declaration. That is, the first port in the component declaration corresponds to the first actual in the component instantiation, the second with the second, and so on.

Example1:

-- Component declaration:

U



component nandg

port (U, V **in** BIT; **W: out** BIT);

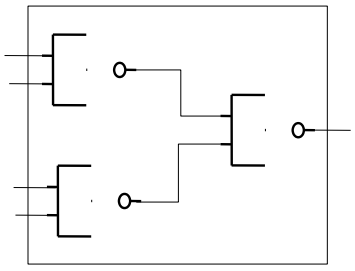
end component;

-- Component instantiation:

G1 : nandg port map (A, B, S1);

G2 : nandg port map (C, D, S2);

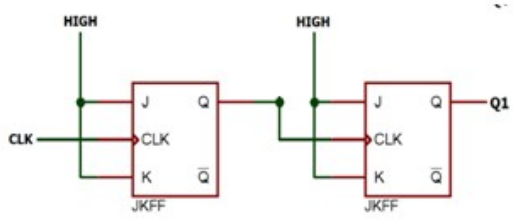
G1 : nandg port map (S1,S2, S3);



G1, G2 & G3 are the component label for the current instantiation of the nandg component. The ordering of the actual is not important. Here S1 is actual and Y is the formal.

If a port in a component instantiation is not connected to the any signal, the keyword OPEN can be used to specify that the port is not connected.

Example 2:

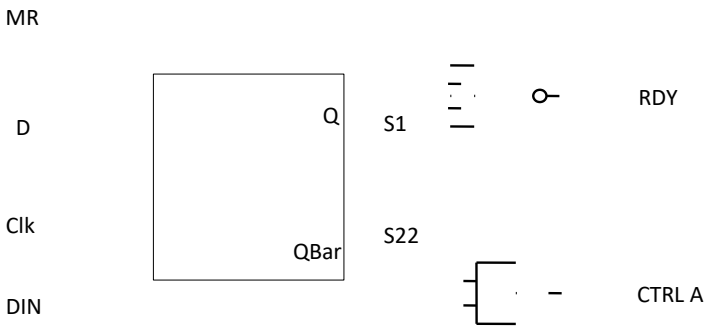


G1 : JKFF port map (1,1,CLK,QA,OPEN);
G2 : JKFF port map (1,1,QA,QB,OPEN);

In named association, an *association-list is* of the form

formal => $actual_1$, $formal_2$ => $actual_2$, ..., $formaal_n$ => $actual_n$

Example:



Component norg
Port (DA,DB: **IN** BIT;
DZ : **OUT**BIT);
End component;

Component andg
Port (x,y : **IN** BIT;
z : **OUT**BIT);
End component;

Component DFF
Port (D,clk : **IN** BIT;
Q, Qbar : **OUT**BIT);
End component;
G1 : DFF portmap (D =>A, clk=>clk, Q=> S1, QBAR => S2);
G2 : norg port map (DA => MR, DB => S1, DZ => RDY);
G3 : Andg port map (x => S2, y => Din , Z => CTRLA);

In this case considering G2, MR is an actual which is declared in the entity port list is associated with the first port (port DA, a formal) of the norg gate, signal ready is associated with the third port (DZ) and S1 is associated with the second port (DB of norg gate.)

- If an actual is a port of mode in, it may not be associated with mode of out or inout.
- If an actual is a port of mode out , it may not be associated with mode of in or inout
- If an actual is a port of mode inout, it may be associated with mode of in, out or inout

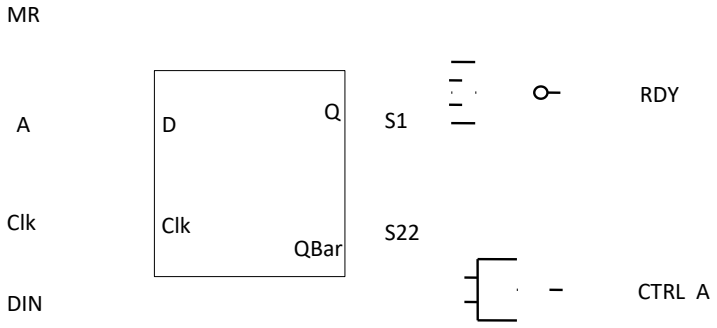
In named association, the ordering of the associations is not important since the mapping between the actuals and formals are explicitly specified.

The signals used to interconnect components can also be

- Slices,

- Vectors, or,
- Array elements.

Write the VHDL code in structural modelling for the following circuit.

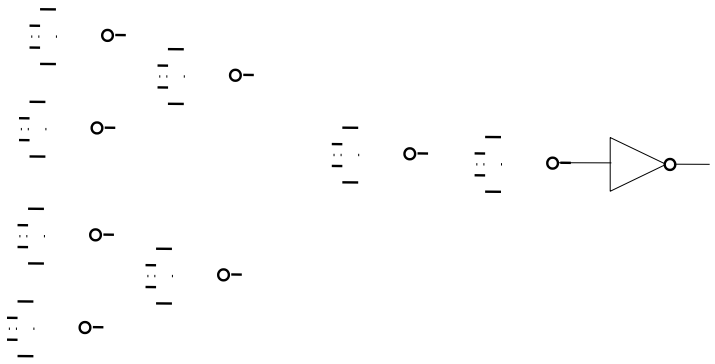



```
entity GATING is
port (A, CK, MR, DIN: in BIT; RDY, CTRLA: out BIT);
end GATING;
architecture STRUCTURE_VIEW of GATING is
        component AND2
        port (X, Y: in BIT; Z: out BIT);
        end component;

        component DFF
        port (D, CLOCK: in BIT; Q, QBAR: out BIT);
        end component;

        component NOR2
        port (A, B: in BIT; Z: out BIT);
        end component;
        signal SI, S2: BIT;
begin
        D1: DFF port map (A, CK, SI, S2);
        A1: AND2 port map (S2, DIN, CTRLA);
        N1: NOR2 port map (SI, MR, RDY);
end STRUCTURE_VIEW;
```

Three components, AND2, DFF, and NOR2, are declared. These components are instantiated in the architecture body via three component instantiation statements, and the instantiated components are connected to each other via signals SI and S2. The component instantiation statements are concurrent statements, and therefore, their order of appearance in the architecture body is not important. A component can, in general, be instantiated any number of times. However, each instantiation must have a unique component label; as an example, A1 is the component label for the AND2 component instantiation.

Other Examples

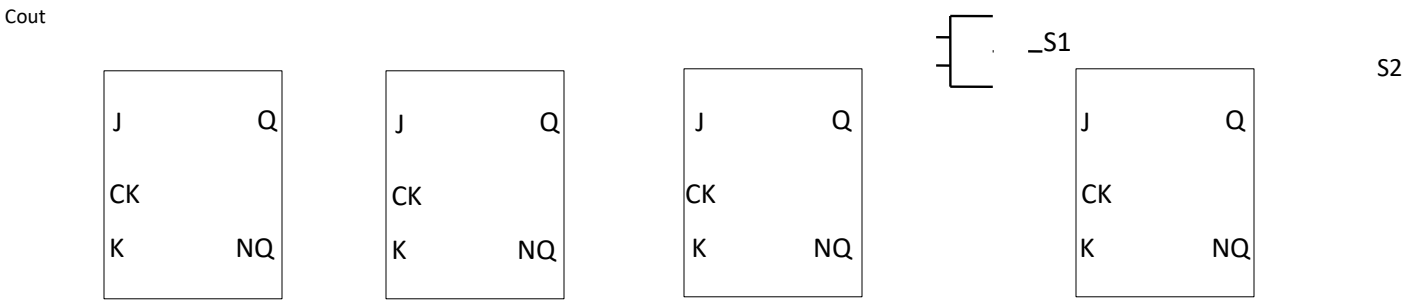
Ex 1: The structural model for a 9-bit parity generator circuit is as shown below.



```
entity PARITY_9_BIT is
port (D: in BIT_VECTOR(8 downto 0); EVEN: out BIT;
        ODD: buffer BIT);
end PARITY_9_BIT;
architecture structural of PARITY_9_BIT is
        component XOR2
        port (A, B: in BIT; Z: out BIT);
        end component;
        component INV2
        port (A: in BIT; Z: out BIT);
        end component;
        signal E0, E1, E2, E3, F0, F1, H0: BIT;
begin
        XE0: XOR2 port map (D(0), D(1), E0);
        XE1: XOR2 port map (D(2), D(3), E1);
        XE2: XOR2 port map (D(4), D(5), E2);
        XE3: XOR2 port map (D(6), D(7), E3);
        XF0: XOR2 port map (E0, E1, F0);
        XF1: XOR2 port map (E2, E3, F1);
        XH0: XOR2 port map (F0, F1, H0);
        XODD: XOR2 port map (H0, D(8), ODD);
        XEVEN: INV2 port map (ODD, EVEN);
end structural;
```

In this example, port ODD is of mode buffer since the value of this port is being read as well as written to inside the architecture.

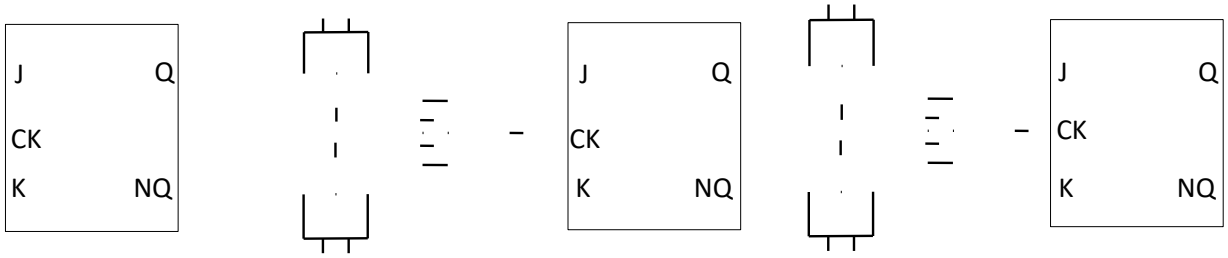
Ex 2: The structural model for decade counter using J-K flip-flops is as shown below.



```
entity decade  is
        port (COUNT: in BIT; Z: buffer BIT_VECTOR(0 to 3));
end decade;
architecture structural  of decade is

        component JK_FF
                port (J, K, CK: in BIT; Q, NQ: buffer BIT);
        end component;
        component andg
                port (A, B: in BIT; C: out BIT);
        end component;
        signal S1, S2: BIT;
        signal S_HIGH: BIT := '1';
begin
        A1: AND_GATE port map (Z(2), Z(1), S1);
        G1: JK_FF port map (1, 1, COUNT, Z(0), open);
        G2: JK_FF port map (S2, 1, Z(0), Z(1), open);
        G3: JK_FF port map (1, 1, Z(1), Z(2), open);
        G4: JK_FF port map (S1, 1, Z(0), Z(3), S2);
end structural;
```

This example illustrates the point that only signals can be used as actuals inside a port map. If a constant, say *'V,* is to be set for one of the ports, as in instance JKI, it is necessary to define a signal, say S_HIGH, that contains this value and then use this signal as an actual for this port. It would be an error to use the constant value directly as an actual in a port map.

Ex 3: Consider the 3-bit up-down counter circuit shown below and its structural model

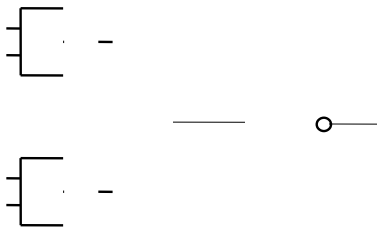



```
entity UP_DOWN is
        port (CLK, CNT_UP, CNT_DOWN: in BIT;
        A, B, C: buffer BIT);
end UP_DOWN;
architecture structural of UP_DOWN is
        component JK_FF
                port (J, K, CK: in BIT; 0, ON: buffer BIT);
        end component;
        component AND2
```

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

    component OR2
        port (A, B: in BIT; C: out BIT);
    end component;
        signal SI, S2, S3, S4, S5, S6, S7, S8: BIT;
        signal ONE: BIT := '1';
    begin
        JK1: JK_FF port map (1, 1, CLK, A, S1);
        A1: AND2 port map (CNT_UP, A, S2);
        A2: AND2 port map (S1, CNT_DOWN, S3);
        O1: OR2 port map (S2, S3, S4);
        JK2: JK_FF port map (1, 1, S4, B, S5);
        A3; AND2 port map (B, CNT_UP, S7);
        A4: AND2 port map (S5, CNT_DOWN, S6);
        O2: OR2 port map (S7, S6, S8);
        JK3: JK_FF port map (1, 1, S8, C, open):
    end structural;
```

Resolving Signal Values

If outputs of two components drive a common signal, then the value of the signal must be resolved using a resolution function. For example, consider the circuit shown below that shows two and gates driving a common signal, RSI, which is inverted to produce the result in Z.



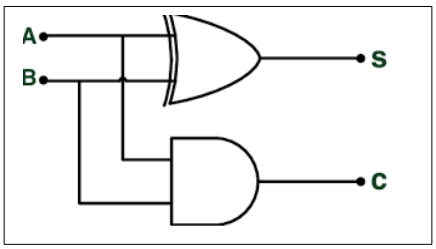
```
entity DRIVING_SIGNAL is
        port (A, B, C, D: in BIT; Z: out BIT);
end DRIVING_SIGNAL;
            {-- PULL_UP is the name of a function defined in package RF_PACK that
              -- has been compiled into the working library}

use WORK.RF_PACK.PULL_UP;

architecture RESOLVED of DRIVING_SIGNAL is
        signal RS1: PULL_UP BIT;
        component AND2
            port (IN1, IN2: in BIT; OUT1: out BIT);
        end component;
        component INV
            port (X: in BIT; Y: out BIT);
        end component;
begin
        A1: AND2 port map (A, B, RS1);
```

```
        A2: AND2 port map (C, D, RS1);
        I1: INV port map (RS1, Z);
    end RESOLVED;
```

The key point here is that even though an assignment to signal RSI is not being made explicitly using signal assignment statements, the signal RSI is being driven by two output ports, and therefore, must be resolved using a resolution function. In the previous example, the PULL_UP resolution function is associated with signal RSI. This implies that the values of the outputs of the and gates are passed through the resolution function before a value is assigned to signal RSI. In general, each out, inout, and buffer port of a component creates a driver for the signal with which it is associated.

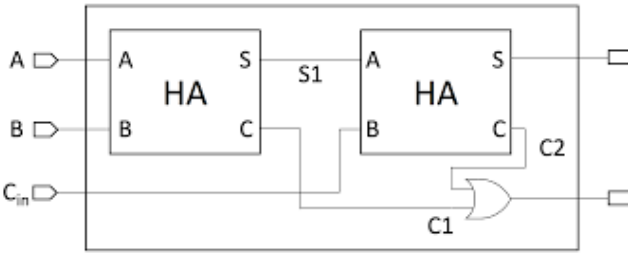
1. Structural model for HALF ADDER

```
Entity HA_S is
   Port ( a, b : in  STD_LOGIC;
          s : out  STD_LOGIC ;
          c : out  STD_LOGIC);
end HA_S;

Architecture Structural of HA_S is

component XORgate
        port (x,y : in STD_LOGIC; z : out STD_LOGIC);
end component;
component ANDgate
        port (x,y : in STD_LOGIC; z : out STD_LOGIC);
end component;

begin
        G1 : XORgate port map (a,b,s);
        G2 : ANDgate port map (a,b,c);
end Structural;
```



2. Structural model for FULL ADDER

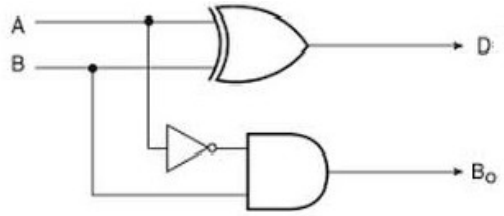
```
Entity FA_S is
   Port ( a  b, c : in  STD_LOGIC;
        s : out  STD_LOGIC;
        Cout : out  STD_LOGIC
end FA_S;
Architecture Structural of FA_S is
component HA
        port (w,x : in STD_LOGIC;
              y,z : out STD_LOGIC);
end component;
component ORgate
        port (x,y : in STD_LOGIC;
              z : out STD_LOGIC);
end component;

signal s1,c1,c2 : STD_LOGIC;
begin
G1 : HA port map (a,b,s1,c1);
G2 : HA port map (s1,c,s,c2);
G3 : ORgate port map (c1,c2,ca);
```



```
end Structural;
```

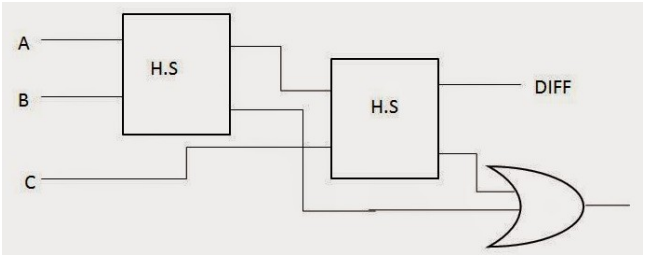
3. Structural model for HALF SUBTRACTOR

```
Entity HS_S is
   Port ( a, b : in  STD_LOGIC;
          d, ba : out  STD_LOGIC);
end HS_S;

architecture Structural of HS_S is
component XORgate
          port (x,y : in STD_LOGIC; z : out STD_LOGIC);
end component;

component NOTgate
          port (x : in STD_LOGIC; y : out STD_LOGIC);
end component;

component ANDgate
          port (x,y : in STD_LOGIC; z : out STD_LOGIC);
end component;

signal x1 : STD_LOGIC;

begin
G1 : XORgate port map (a,b,d);
G2 : NOTgate port map (a,x1);
G3 : ANDgate port map (x1,b,ba);

end Structural;
```



4. Structural model for FULL SUBTRACTOR

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity FS_S is
   Port ( a : in  STD_LOGIC;
          b : in  STD_LOGIC;
          c : in  STD_LOGIC;
          d : out  STD_LOGIC;
          ba : out  STD_LOGIC);
end FS_S;

architecture Structural of FS_S is
component HS
          port (w,x :in STD_LOGIC; y,z : out STD_LOGIC);
```



```
end component;

component ORgate
        port (x,y :in STD_LOGIC; z : out STD_LOGIC);
end component;

signal ba1,ba2,d1 : STD_LOGIC;

begin

G1 : HS port map (a,b,d1,ba1);
G2 : HS port map (d1,c,d,ba2);
G3 : ORgate port map (ba1,ba2,ba);

end Structural;
```