UNIT-1

Hardware Description Languages

VHDL :-

V → Very High Speed Integrated Circuits.

H → Hardware

D → Description

L → Language.

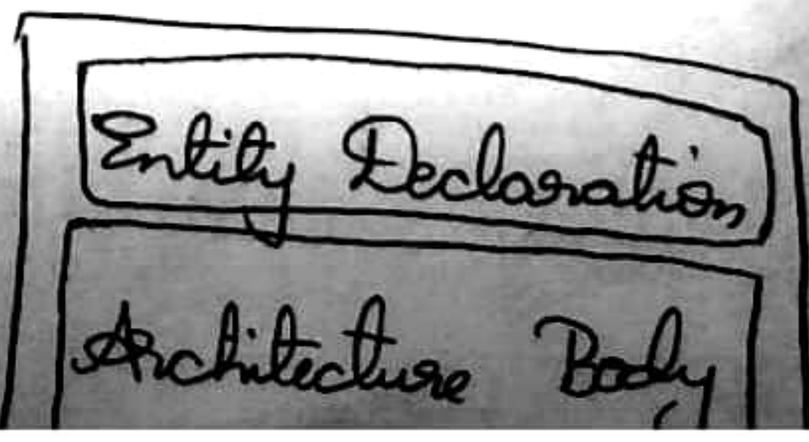
* A Hardware Description Language (HDL) can be used to model a digital System.
* To describe an entity, VHDL provides five different types of primary constructs. They are:

1) Entity Declaration
2) Architecture Body
3) Configuration Declaration
4) Package Declaration
5) Package Body.

* A basic programming Structure of VHDL is consisting of

→ Entity Declaration

→ Architecture Body.

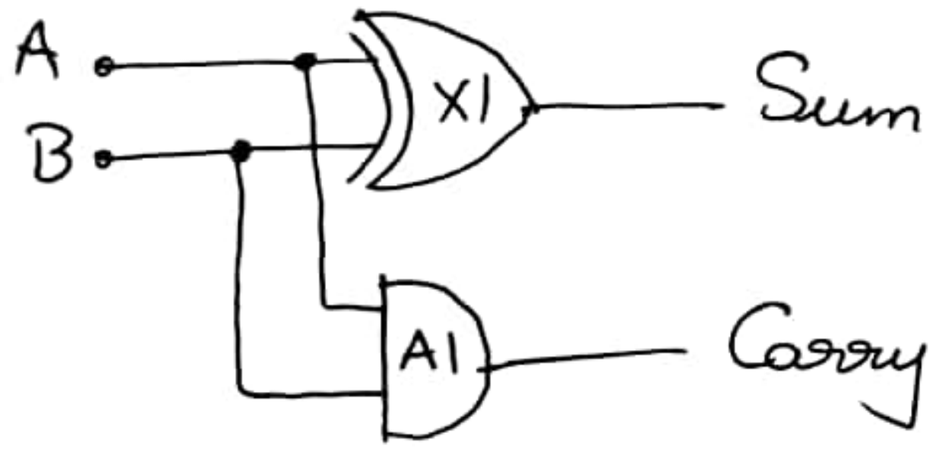


Entity Declaration :-

The entity declaration specifies the name of the entity being modeled and lists the set of interface ports.

* Ports are signals through which the inputs and outputs are connected to entity.

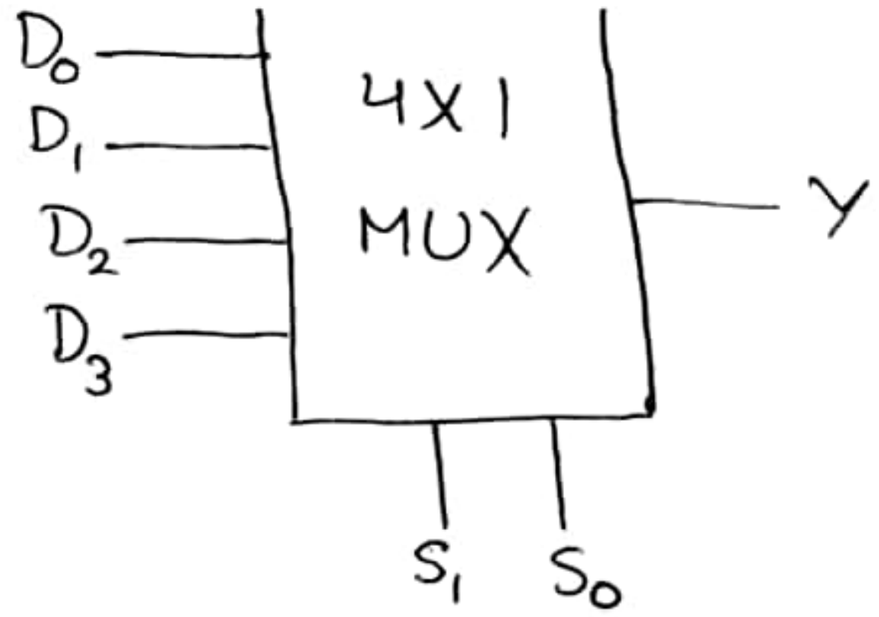
* Here, is an example of an entity declaration for half adder circuit.




```
* entity Half_Adder is
    port (A, B : in STD_LOGIC;
          Sum, Carry : out STD_LOGIC);
end Half_Adder;
```

* Here the entity Half_Adder has two input ports (A and B) and two output ports (Sum and Carry).

* The following is another example of an entity declaration for 4X1 Multiplexer.




```
* entity MUX_4X1 is
    port (D: in STD_LOGIC_VECTOR(0 to 3);
          S: in STD_LOGIC_VECTOR(1 downto 0);
          Y: out STD_LOGIC);
end MUX_4X1;
```

Architecture Body :-

The internal details of an entity are specified by an architecture body using any of the following modelling styles :-

1) Structural Modeling → As a set of inter-connected components.
2) Dataflow Modeling → As a set of concurrent signal assignment statements.
3) Behavioral Modeling → As a set of sequential signal assignment statements.

* He can also describe the architecture as combination of above three modelings.

* Structural Style of Modeling :-

In the Structural Style of modeling, an entity is described as a set of interconnected components.

* The following is the example for Half_Adder.

```
* entity Half_Adder is
    port (A,B : in STD_LOGIC;
          Sum, Carry : out STD_LOGIC);
end Half_Adder;
architecture Structural of Half_Adder is
  component XOR2
    port (X,Y : in STD_LOGIC;
          Z   : out STD_LOGIC);
  end component;
  component AND2
    port (P,Q : in STD_LOGIC;
          R   : out STD_LOGIC);
  end component;
begin
  X1 : XOR2 port map (A, B, Sum);
  A1 : AND2 port map (A, B, Carry);
end Structural;
```

* The required components for Half Adder are XOR and AND gates.

* In general, the architecture body is composed of two parts :-
  i) Declarative part (before the keyword begin)
  ii) Statement part (after the keyword begin).
* The required components should be declared in declarative part before using in architecture.
* The programs for components are given as follows :-

2-input XOR Gate :-

```
entity XOR2 is
    port (X, Y : in STD_LOGIC;
          Z : out STD_LOGIC);
end XOR2;
architecture XOR2 of XOR2 is
begin
    Z <= X xor Y;
end XOR2;
```

2-input AND Gate :-

```
entity AND2 is
    port (P, Q : in STD_LOGIC;
          R : out STD_LOGIC);
end AND2;
architecture AND2 of AND2 is
begin
    R <= P and Q;
end AND2;
```

* Dataflow Style of modeling:-

In this Style of modeling, an entity is described using concurrent Signal assigument Statements.

* The following is the example for Half Adder.

```
* entity Half_Adder is
    port (A,B: in STD_LOGIC;
          Sum, Carry: out STD_LOGIC);
  end Half_Adder;
  architecture Dataflow of Half_Adder is
  begin
     Sum <= A xor B after 8 ns;
     Carry <= A and B after 4ns;
  end Dataflow;
```

* In Signal assignment Statement, the Symbol <= implies an assigument of value to a Signal.

* Delay information is included in the Signal assignment Statements using "after" clauses.

* In the above given Half Adder example, both Signal assigument Statements execute concurrently.

* Behavioral Style of Modeling :-

In this type of modeling, an entity is ~~dest~~ described as a set of sequential signal assignment statements.

* Here the behavior of an entity is described which are executed sequentially.

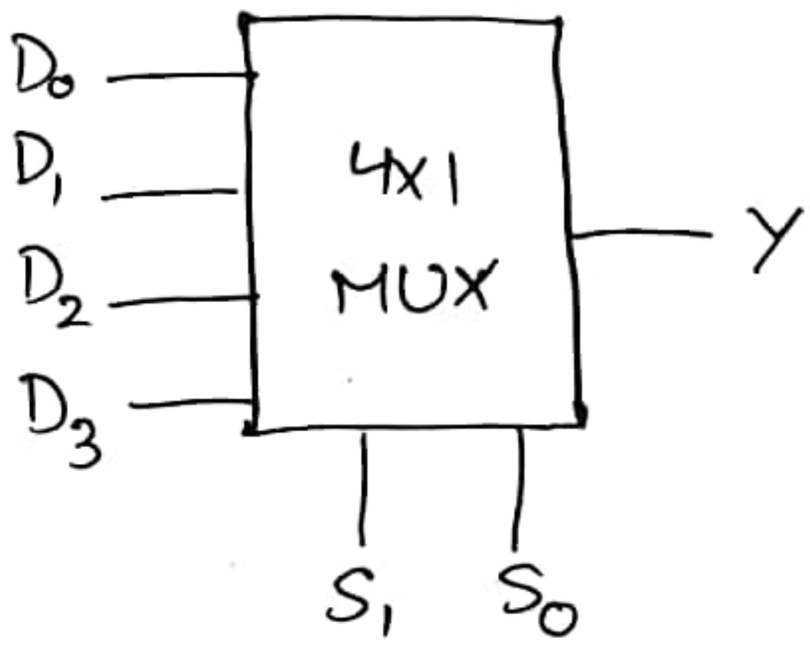
* Half Adder Example ⇒

```
* entity Half_Adder is
    port (A,B : in STD_LOGIC;
          Sum, Carry : out STD_LOGIC);
  end Half_Adder;
architecture Behavioral of Half_Adder is
begin
process (A, B)
begin
   if (A = '0' and B = '0') then
        Sum <= 0;  Carry <= '0';
   elsif (A = '0' and B = '1') then
        Sum <= '1';  Carry <= '0';
   elsif (A = '1' and B = '0') then
        Sum <= '1';  Carry <= '0';
   else
        Sum <= '0';  Carry <= '1';
   end if;
```

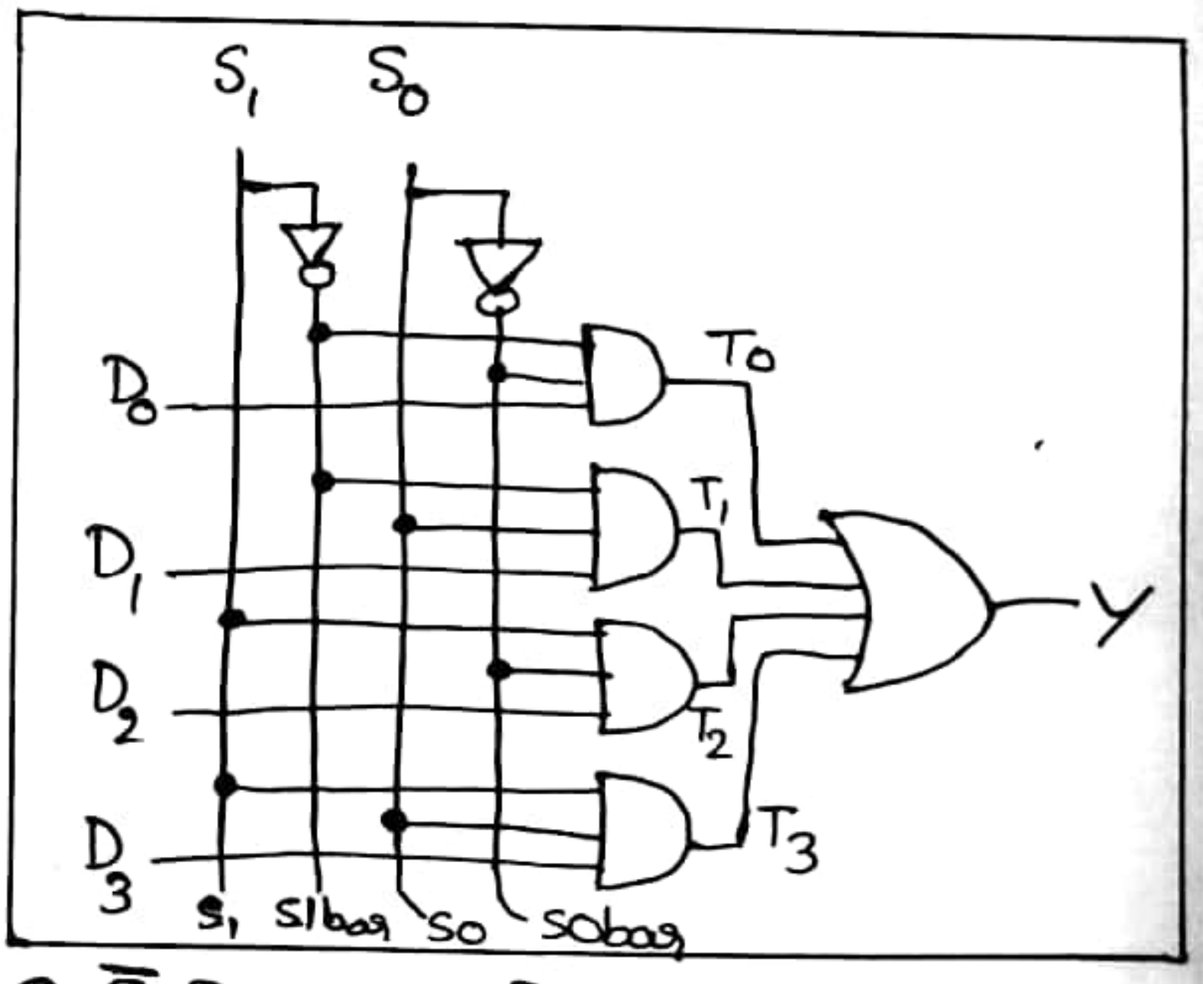
```
end process;
end Behavioral;
```

* The Statements appearing inside a process Statement are Sequential Statements and are executed Sequentially.
* A process Statement constitutes a Sensitivity list, and the process Statement is invoked whenever there is an event on any Signal in this list.

* A VHDL program for 4X1 Multiplexer :-



S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3



$$Y = \bar{S}_1 \bar{S}_0 D_0 + \bar{S}_1 S_0 D_1 + S_1 \bar{S}_0 D_2 + S_1 S_0 D_3 .$$

4X1 MUX using Dataflow Modeling :-

```
entity MUX_4X1 is
  port (D : in STD_LOGIC_VECTOR (0 to 3));
        S : in STD_LOGIC_VECTOR (1 downto 0));
        Y : out STD_LOGIC);
end MUX_4X1;
architecture Dataflow of MUX_4X1 is
begin
  Y <= (not (S(1)) and not (S(0)) and D(0)) or
       (not (S(1)) and  S(0) and D(1)) or
       (S(1) and not (S(0)) and D(2)) or
       (S(1) and S(0) and D(3));
end Dataflow;
```

Architecture of 4X1 MUX using Behavioral :-

```
architecture Behavioral of MUX_4X1 is
begin
process (D,S)
begin
  if (S = "00") then Y <= D(0);
  elsif (S = "01") then Y <= D(1);
  elsif (S = "10") then Y <= D(2);
  else Y <= D(3);
end Behavioral;
```

Architecture for 4X1 MUX using Structural :-

```
architecture Structural of MUX_4X1 is
component NOTG
  port (x : in STD_LOGIC;
        y : out STD_LOGIC);
end component;
component ANDG
  port (X,Y : in STD_LOGIC;
        z : in STD_LOGIC;
        out : out STD_LOGIC);
end component;
component ORG
  port (A,B,C,D : in STD_LOGIC;
        Y : out STD_LOGIC);
end component;
    signal S1bar, S0bar : STD_LOGIC;
begin
    signal T : STD_LOGIC_VECTOR(0 to 3);
MUX: begin
  N1 : NOTG port map (S(1), S1bar);
  N2 : NOTG port map (S(0), S0bar);
  A1 : ANDG port map (S1bar, S0bar, D(0), T(0));
  A2 : ANDG port map (S1bar, S(0), D(1), T(1));
  A3 : ANDG port map (S(1), S0bar, D(2), T(2));
  A4 : ANDG port map (S(1), S(0), D(3), T(3));
  OR1 : ORG port map (T(0), T(1), T(2), T(3), Y);
Structural;
```

Ex:- Write a VHDL program for 8x1 Multiplexer using Behavioral Modeling. (Include Enable pin)

Sol:- The truth table for 8x1 Multiplexer with Enable pin is as shown below :-

EN	S_2	S_1	S_0	Y
0	x	x	x	0
1	0	0	0	D_0
1	0	0	1	D_1
1	0	1	0	D_2
1	0	1	1	D_3
1	1	0	0	D_4
1	1	0	1	D_5
1	1	1	0	D_6
1	1	1	1	D_7

No. of inputs = 8

No. of select lines = 3

No. of outputs = 1

*VHDL program using Behavioral Modeling.

```
entity MUX_8X1 is
    port ( D : in STD_LOGIC_VECTOR(0 to 7);
           S : in STD_LOGIC_VECTOR (2 downto 0);
           EN : in STD_LOGIC;
           Y : out STD_LOGIC);
end MUX_8X1;
```

```
architecture Behavioral of MUX_8X1 is
begin
process (EN, D, S)
begin
   if S= "00 if EN = '0' then Y <= 0 ;
   else
   if (S = "000") then  Y <= D(0);
   elsif (S = "001") then  Y <= D(1);
   elsif (S = "010") then  Y <= D(2);
   elsif (S = "011") then Y <= D(3);
   elsif (S = "100") then Y <= D(4);
   elsif (S = "101") then Y <= D(5);
   elsif (S = "110") then Y <= D(6);
   else   Y <= D(7);
   end if;
 end if;
end Process;
end Behavioral;
```

*. It is possible to mix the three modeling styles that we have seen so far in a single architecture body.

* Within an architecture body, we can use component instantiation statements (that represent structure), concurrent signal assignment statements (that represent dataflow) and Process statements (that represent behavior).

* Configuration Declaration :- It is used to select one of possibly many architecture bodies than an entity may have.

```
configuration configuration_name of entity_name is
  -- configuration declarations
  for architecture_name
    for instance_label:component_name
      use
entity library_name.entity_name(arch_name);
    end for;
  -- other for clauses
  end for;
end [configuration] [configuration_name];
```

```
library CMOS_LIB, MY_LIB;
configuration HA_BINDING of HALF_ADDER is
  for HA_STRUCTURE
    for X1: XOR2
      use entity CMOS_LIB.XOR_GATE(DATAFLOW);
    end for;
    for A1: AND2
      use configuration MY_LIB.AND_CONFIG;
    end for;
  end for;
end HA_BINDING;
```

* Package Declaration :-

It is used to store a set of common declarations, such as components, Types, Procedures and functions.

Syntax :-

```
package package_name is
package_declarations
end package package_name;
```

Example :-

```
package EXAMPLE_PACK is
  type SUMMER is (MAY, JUN, JUL, AUG, SEP);
  component D_FLIP_FLOP
    port (D, CK: in BIT; Q, QBAR: out BIT);
  end component;
  constant PIN2PIN_DELAY: TIME := 125 ns;
  function INT2BIT_VEC (INT_VALUE: INTEGER)
    return BIT_VECTOR;
end EXAMPLE_PACK;
```

* These declarations can be imported into other design units using a "USE" clause.

* Package Body :-

```
package body package_name is
        declarations
        deferred constant declarations
        subprogram bodies
end package_name;
```

```
package body EXAMPLE_PACK is
  function INT2BIT_VEC (INT_VALUE: INTEGER)
    return BIT_VECTOR is
  begin
    -- Behavior of function described here.
  end INT2BIT_VEC;
end EXAMPLE_PACK;
```

→ A package body is used to store the definitions of functions and procedures that were declared in the corresponding package declaration, and also the complete constant declarations.

* Elements of VHDL :-

The following are the basic elements of VHDL language.

1) Identifiers
2) Data Objects
3) Data Types.
4) Operators.

Identifiers :-

* There are two kinds of identifiers in VHDL

    1) Basic Identifiers
    2) Extended Identifiers.

* The language defines a set of reserved words; these words also called "Keywords", have a specific meaning in the language and therefore cannot be used as identifiers.

Basic Identifier :-

A Basic identifiers in VHDL is composed of a sequence of one or more characters.

→ upper case letters (A to Z)
→ Lower case letters (a to z)
→ A digit (0 to 9)
→ Underscore (_).

* The first character in a basic identifier must be a letter and last character in a basic identifier maynot be an underscore.

* Also two underscore characters may cannot appear consecutively.

Example :- 1) Count, COUNT, CoUnT → all Same
2) DRIVE_BUS, CONST32_59, 82d2.

Extended Identifiers :-

An extended identifier is a sequence of characters written between two backslashes.

* Any of the allowable characters can be used including ! @ $ ...
* Within an extended identifier, lower-case and upper case letters are considered to be distinct.

Example :- 1) \Count\ -- distinct from \COUNT\

2) \TEST\ , \2FOR$\, \~Q\.

Data Objects :-

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

* There are four different types of data objects
  1) Constant
  2) Variable
  3) Signal
  4) File.

Ex :- variable COUNT : INTEGER

Constant Declarations :-

A constant can hold a single value of a given type. This value is assigned to the constant before simulation ~~state~~ starts and the value cannot be changed during the course of the simulation.

Ex:- 1) constant RISE_TIME : TIME := 10ns;

2) constant BUS-WIDTH : INTEGER := 8 ;

* The value of the constant has not been specified in this case. Such a constant is called a deferred constant, and it can appear only inside a package declaration.

Variable Declarations :-

A variable can hold a single value of a given type. However, in this case, different values can be assigned to the variable at different times using variable assignment statement (:=).

Ex:- 1) variable CTRL : BIT_VECTOR (0 to 10);

2) variable Sum : INTEGER range 0 to 100;

3) variable FOUND, DONE : BOOLEAN;

Signal Declarations:-

A signal can hold a list of values, which include the current value of the signal and a set of future values that are to appear on the signal.

Eg:- 1) signal clock : BIT;

2) signal Data_Bus : BIT_VECTOR(0 to 7);

3) signal INIT_P : STD_LOGIC;

File Declaration:-

A file contains a sequence of values. Values can be read or written to the file using read procedures and write procedures.

* A file is declared using a file declaration. The syntax for file declaration is:

file file_names : file_type_name [[open mode] is string_expression];

* The string expression is interpreted by the host environment as the physical name of a file.

The mode specifies whether the file is to be

used as a read-only or write-only, or in the append mode.

Ex:- 1) file STIMULUS : TEXT open READ_MODE is "/usr/home/jb/add.sti";

2) file VECTORS : BIT_FILE is "/usr/home/james/add.ved";

Data Types :-

* Every data object in VHDL can hold a value that belongs to a set of values.
* This set of values is specified by using a type declaration.
* The Predefined VHDL data types are :-

→ Bit	→ Integer
→ Bit_vector	→ Real
→ Boolean	→ Time
→ character	→ String.

Integer :- The type integer is defined as the range of integers including $-(2^{31}-1)$ to $+(2^{31}-1)$: VHDL implementations may extend this range.

Ex:- Signal X : integer;
Signal Y : integer := 67;

Boolean :- Type boolean has two values, TRUE and FALSE. The character type contains all of the characters in the ISO 8-bit character set the first 128 are the ASCII characters.

* Built-in operators for the integer and boolean types are listed in above table.

Ex:- signal L: Boolean;
signal N, M: Boolean;

Bit Type :- Since VHDL is used to model digital systems, it is useful to have a data type to represent bit values. The predefined enumeration type bit serves this purpose. It is defined as :-

Type bit is ('0', '1');

* Signals and variables of type bit can have values 0 and 1;

Ex:- signal X: Bit;
signal Y: Bit := "1";

String :- The type String is a vector of elements of the type character.

* The way the vector elements are indexed depends on the defined range - either ascending or descending.

Ex:- constant Message1 : String (1 to 19) :=
"hold time violation";
signal Letter1 : character;
signal Message2 : String (1 to 10);

Message2 <= "Net" & Letter1;

* Here & is a concatenation operator.

Real:- Apart from the Standard types like integer and STD-LOGIC_VECTOR's VHDL also offer Real data types. But a real data type has a big disadvantage. It is not Synthesizable. It can be used only for Simulation purposes.

* The real data type is defined in the library called MATH_REAL. So we have to include the following line before the entity declaration in the code.

```
use IEEE.MATH_REAL.all;
```

Ex:- 1) Signal X : real := -MATH_PI/3.0;

↙ A real variable X, initialized to $\frac{\pi}{3}$ (60°).

2) Signal max, min, power1, exp_result : real := 0.0;

Data Operators:-

The predefined operators in the language are classified into the following six categories:-

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

Logical Operators:-

There are 7-logical operators corresponding to 7-logic gates ⇒

* and * xor * nand * not.
* or * xnor * nor

→ During evaluation of logical operators, bit values 0 and 1 are treated as FALSE and TRUE values of the BOOLEAN type, respectively.

Relational Operators:-

These are given as

* = (Equality)
* /= (Inequality)
* > (Greater than)
* < (Less than)

* >= (Greater than or equal to)
* <= (Less than or equal to).

Ex:- 1) "VHDL" < "VHDL92"

2) MVL'('U') < MVL'('Z').

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

Shift Operators:-

* SLL → Shift Left } Left-over bits filled
* SRL → Shift Right } with '0'.
* SLA → Shift Left Arithemetic
* SRA → Shift Right Arithmetic
* ROR → Rotate Right
* ROL → Rotate Left.

Ex:- "1001010" → It is a BIT_VECTOR.

1) sll 2 is "0101000"
2) srl 3 is "0001001"
3) sla 2 is "0101000"
-- filled with right-most bit.
4) sra 3 is "1111001"
-- filled with left most bit.
5) ror 3 is "0101001"
6) rol 2 is "0101010".

Adding operators :-

+	-	&

* The operands for the + (addition) and - (subtraction) operators must be of the same numeric type.
* The operands for the & (concatenation) operator can be either one-dimensional array type.

Ex:- '0' & '1' results in "01".

'C' & 'A' & 'T' results in "CAT".

"BA" & "LL" results in "BALL".

Multiplying Operators :-

*	mod	rem

* The * (multiplication) and / (division) operators are predefined for both integer & floating point type.

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

A mod B = A - B * N -- for some integer.

Miscellaneous Operators :-

abs	**

→ abs (Absolute)

→ ** (Exponentiation)

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)] [is]
    [process-item-declarations]
begin
  Sequential-Statements ; These are -->
      variable-assignment-Statement
      Signal-assignment-Statement
    wait-Statement
      if-Statement
      case-Statement
      loop-Statement
      null-Statement
      ---
end process [proce
```

* A set of Signals to which the process is sensitive is defined by the Sensitivity List.
* Each time, an event occurs on any of the Signals in the Sensitivity list, then the Sequential Statements within the process are executed in a Sequential order.
* The process then Suspends after executing the last Sequential Statement and waits for another event to occur on a Signal in the Sensitivity list.

Variable assignment Statement :-

Variables can be declared and used inside a process Statement. A variable is assigned a value using the variable assignment Statement.

```
variable_object := Expression
```

Ex:-

```
process (A)
    variable EVENTS_ON_A : INTEGER := -1;
begin
    EVENTS_ON_A := EVENTS_ON_A +1;
end process;
```

Signal Assignment Statement :-

Signals are assigned values using a Signal assignment statement. The simplest form of a Signal assignment is :

```
signal-object <= Expression [after delay-value];
```

* A Signal assignment statement can appear within a process & outside a process.
* If the Signal assignment appear outside of a process, it is considered to be a concurrent Signal assignment statement.
* If the Signal assignment appear within the process, it is considered to be a Sequential Signal assignment statement.

Exe:- 1) COUNTER <= COUNTER + "0010";

2) PAR <= PAR & DIN after 12 ns;

3) Z <= (A0 and A1) & (B0 and B1) after 6ns;

Wait Statement :-

A process Statement may be Suspended by means of a Sensitivity list.

* The wait Statement provides an alternative way to Suspend the execution of a process.

* There are three basic forms of the wait statement.

```
wait on Sensitivity-list ;
wait until boolean-expression;
wait for time-expression;
```

* They may also be combined in a single wait Statement

wait on Sensitivity-list until Boolean-exp for time-exp;

Ex:- wait on A, B, C.
wait until A=B;
wait for 10 ns;
wait on CLOCK for 20 ns;
wait until SUM>100 for 50 ns;
wait on CLOCK until SUM>100;

Ex:-
```
process  -- Sensitivity is not given here.
begin
    wait on DATA;  -- Replaces the Sensitivity list
    SIG_A <= DATA;
    wait for 0 ns;
    SIG_B <= SIG_A;
end process.
```

* wait for 0 → It means to wait for one delta cycle.

IF Statement :-

An if Statement selects a sequence of Statements for execution based on the value of a condition.

* The condition can be any expression that evaluates to a boolean value.

Syntax :- if boolean-expression then
    sequential-Statements.
[elsif boolean-expression then
    sequential-Statements]
[else
    sequential-Statements]
    end if;

* The if Statement is executed by checking each condition sequentially until the first TRUE condition is found; then, the Set of Sequen-tial Statements associated with this condition is executed.
* The if Statement can have zero or more elsif clauses and an optional else clause.

Eg:- if (A nor B) = '1' then
    Z <= '1' after RISE_TIME;
else
    Z <= '0' after FALL_TIME;
end if;

Case Statement:-

The Syntax for case Statement is:

* case expression is
  when choices => Sequential-Statements
  when choices => Sequential-Statements
  -- can have any number of branches.
  [when others => Sequential-Statements]
  end case;

* The CASE Statement Selects one of the branches for execution based on the value of the expression.
* The expression value must be of a discrete type or of a one-dimensional array type.

Ex:- A model for a 4X1 multiplexer using a case Statement is shown here.

```
entity MUX is
    port (A,B,C,D : in STD_LOGIC;
          S       : in STD_LOGIC_VECTOR(0 to 1);
          Z       : out STD_LOGIC);
end MUX;
```

```
architecture MUX_BEHAVIOR of MUX is
   constant MUX_DELAY : TIME := 10 ns;
begin
  PMUX : process (A, B, C, D, S)
      variable TEMP : STD_LOGIC;
  begin
    case S is
      when "00" => TEMP := A;
      when "01" => TEMP := B;
      when "10" => TEMP := C;
      when "11" => TEMP := D;
      when others => TEMP := 'x';
end case;
    Z <= TEMP after MUX_DELAY;
  end process PMUX;
end MUX_BEHAVIOR;
```

Null Statement :-

The NULL Statement is a Sequential Statement that doesnot cause any action to take place; execution continues with the next Statement.

```
null;
```

* For NULL Statement, no action near is performed.

Loop Statement :-

* A loop statement is used to iterate through a set of sequential statements.
* 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.
  i) for ii) while iii) Loop

Ex:- 1)
```
FACTORIAL := 1;
    for Number in 2 to N loop
        FACTORIAL := FACTORIAL * Number;
    end loop;
```

2)
```
J := 0; SUM := 10;
WH_LOOP : while J < 20 loop
    SUM := SUM * 2;
    J := J+3;
end loop;
```

3)
```
SUM := 1; J := 0;
L2 : loop
    J := J+21;
    SUM := SUM * 10;
    exit when SUM>100; end loop L2;
```