Lecture 1

Introduction to VHDL

VHDL: VHSIC (Very High Speed Integrated Circuits) Hardware Description Language, an industry standard language used to model a digital system at many levels of abstraction ranging from the algorithmic level to the gate level.

- VHDL resulted from work done in the '70s and early '80s by the U.S. Department of Defense
- Originated from ADA Language (**ADA** is named after Augusta Ada Byron (1815-52), daughter of Lord Byron, and Countess of Lovelace. She helped Charles Babbage develop programs for the *analytic engine,* the first mechanical computer. She is considered by many to be the world's first programmer)
- In December 1987, VHDL was proposed as an IEEE (Institute of Electrical and Electronics Engineers) standard known as the IEEE Std 1076-1987.
- The VHDL language can be regarded as an integrated amalgamation of the following languages:
  Sequential language +
  Concurrent language +
  Net-List language +
  Timing Specifications +
  Waveform Generation language => VHDL → Case-Insensitive
- The language has constructs that enable you to express the concurrent or sequential behavior of a digital system with or without timing.

VHDL Terms

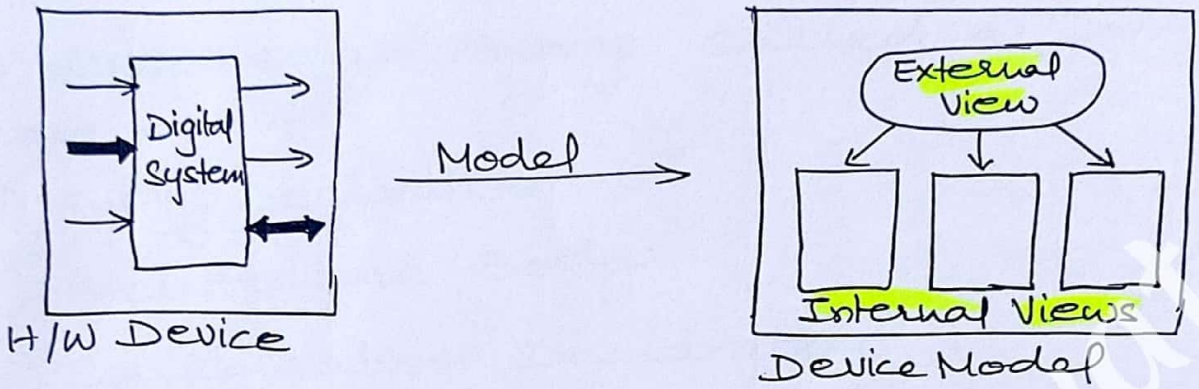
- **ENTITY:** All designs are expressed in terms of entities. An entity is the most basic building block in a design. The uppermost level of the design is the top-level entity. If the design is hierarchical, then the top-level description will have lower-level descriptions contained in it. These lower-level descriptions will be lower-level entities contained in the top-level entity description.
- **ARCHITECTURE:** All entities that can be simulated have an architecture description. The architecture describes the behavior of the entity. The internal details of an entity are specified by an architecture body using any of the following modeling styles:

  1. As a set of interconnected components (to represent **Structure**),

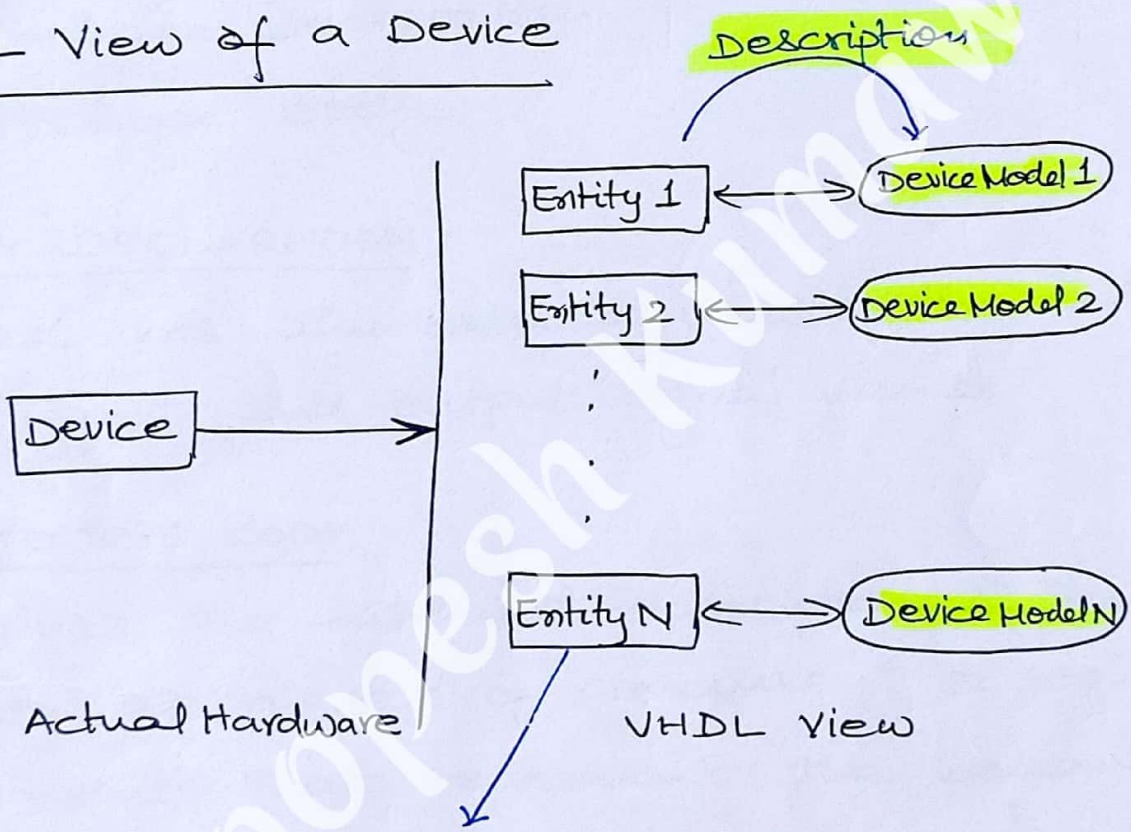
2. As a set of concurrent assignment statements (to represent **Dataflow**),
3. As a set of sequential assignment statements (to represent **Behavior**),
4. Any combination of the above three.

- **CONFIGURATION:** A configuration statement is used to bind a component instance to an entity-architecture pair. A configuration can be considered like a parts list for a design. It describes which behavior to use for each entity, much like a parts list describes which part to use for each part in the design.
- **PACKAGE:** A package is a collection of commonly used data types and subprograms used in a design. Think of a package as a toolbox that contains tools used to build designs. It is used to store a set of common declarations like components, types, procedures, and functions.
- **DRIVER:** This is a source of a signal. If a signal is driven by two sources, then when both sources are active, the signal will have two drivers.
- **BUS:** The term "bus" usually brings to mind a group of signals or a particular method of communication used in the design of hardware. In VHDL, a bus is a special kind of signal that may have its drivers turned off
- **ATTRIBUTE:** An attribute is data that are attached to VHDL objects or predefined data about VHDL objects. Examples are the current drive capability of a buffer or the maximum operating temperature of the device.
- **GENERIC:** A generic is VHDL's term for a parameter that passes information to an entity. For instance, if an entity is a gate level model with a rise and a fall delay, values for the rise and fall delays could be passed into the entity with generics.
- **PROCESS:** A process is the basic unit of execution in VHDL. All operations that are performed in a simulation of a VHDL description are broken into single or multiple processes.

Device Model



VHDL View of a Device



Entity is H/W abstraction of actual H/W device wherein each device model contains one external view and one or more internal views.

NOTE: When an entity X is used in another entity Y, then it becomes a Component for the entity Y. Thus, depending upon level of the model, a component is also an Entity.

ENTITY DESCRIPTION

In VHDL, five different constructs are there to describe an Entity, called as Design Units. These are :-

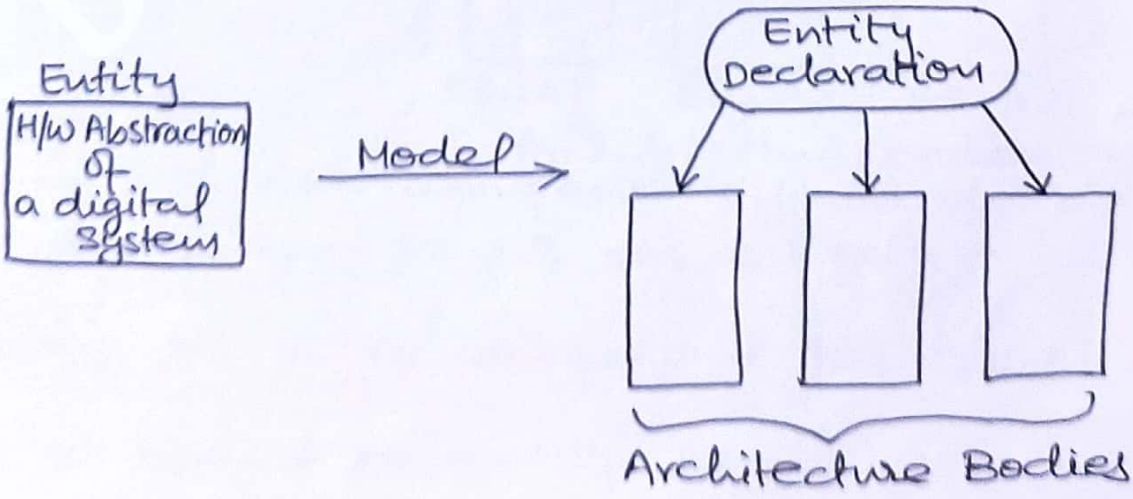
(i) Entity Declaration
(ii) Architecture Body
(iii) Configuration Declaration
(iv) Package Declaration
(v) Package Body

ENTITY DECLARATION

- It describes the external view of the entity. e.g. input and output signal names.

ARCHITECTURE BODY

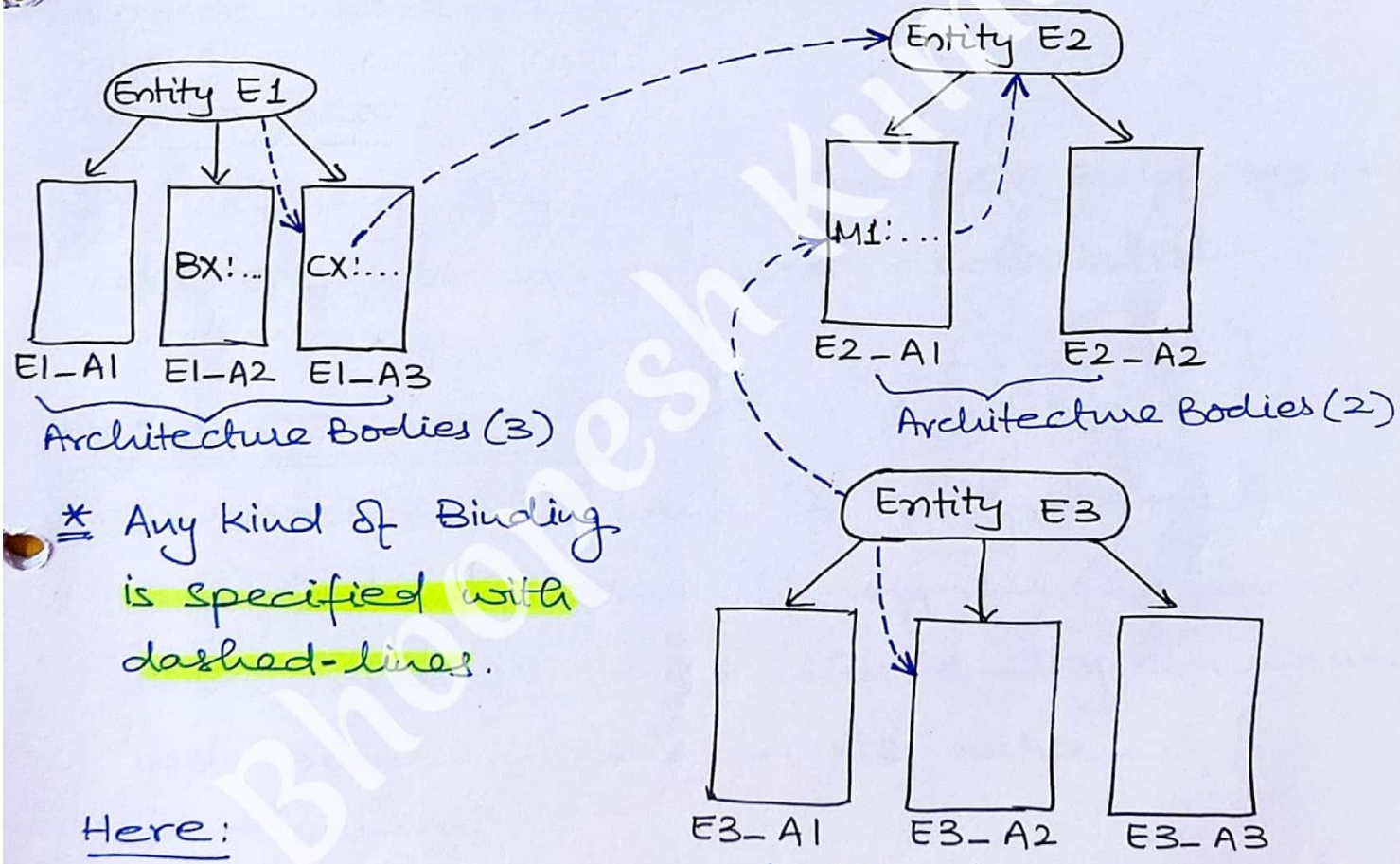
- It gives the internal description of the entity, as a set of concurrent or sequential statements that represents the behavior of the entity.



CONFIGURATION DECLARATION

- Used to create a configuration for an Entity that specifies the binding of one architecture body from many architecture bodies that may be associated with the entity or with different entities or with different components within or outside the entity.

Example:



* Any kind of Binding is specified with dashed-lines.

Here:

- BX and CX are components for architecture body E1_A2 and E1_A3 respectively.
- Similarly, M1 is a component for E2_A1
- E1_A3 is bound to entity E1.
- E2_A1 is bound to entity E2

- Component M1 in architecture body E2_A1 is bound to entity E3
- Component CX in E1_A3 is bound to entity E2.

PACKAGE DECLARATION

A package declaration encapsulates a set of related declarations, such as type declarations, subtype declarations, and subprogram-declarations, which can be shared across two or more design units.

PACKAGE BODY

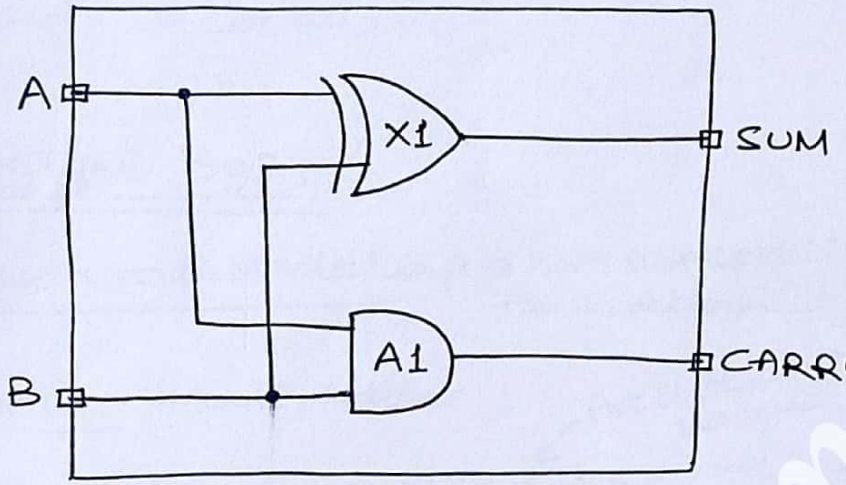
It contains the definitions of subprograms declared in a package declaration.

ENTITY DECLARATION

- It specifies the name of entity being modeled and lists the set of signals (known as Ports) through which the entity communicates with other models in its external environment.
- It does not specify anything about the internals of the entity.

EXAMPLES

(1.) HALF-ADDER



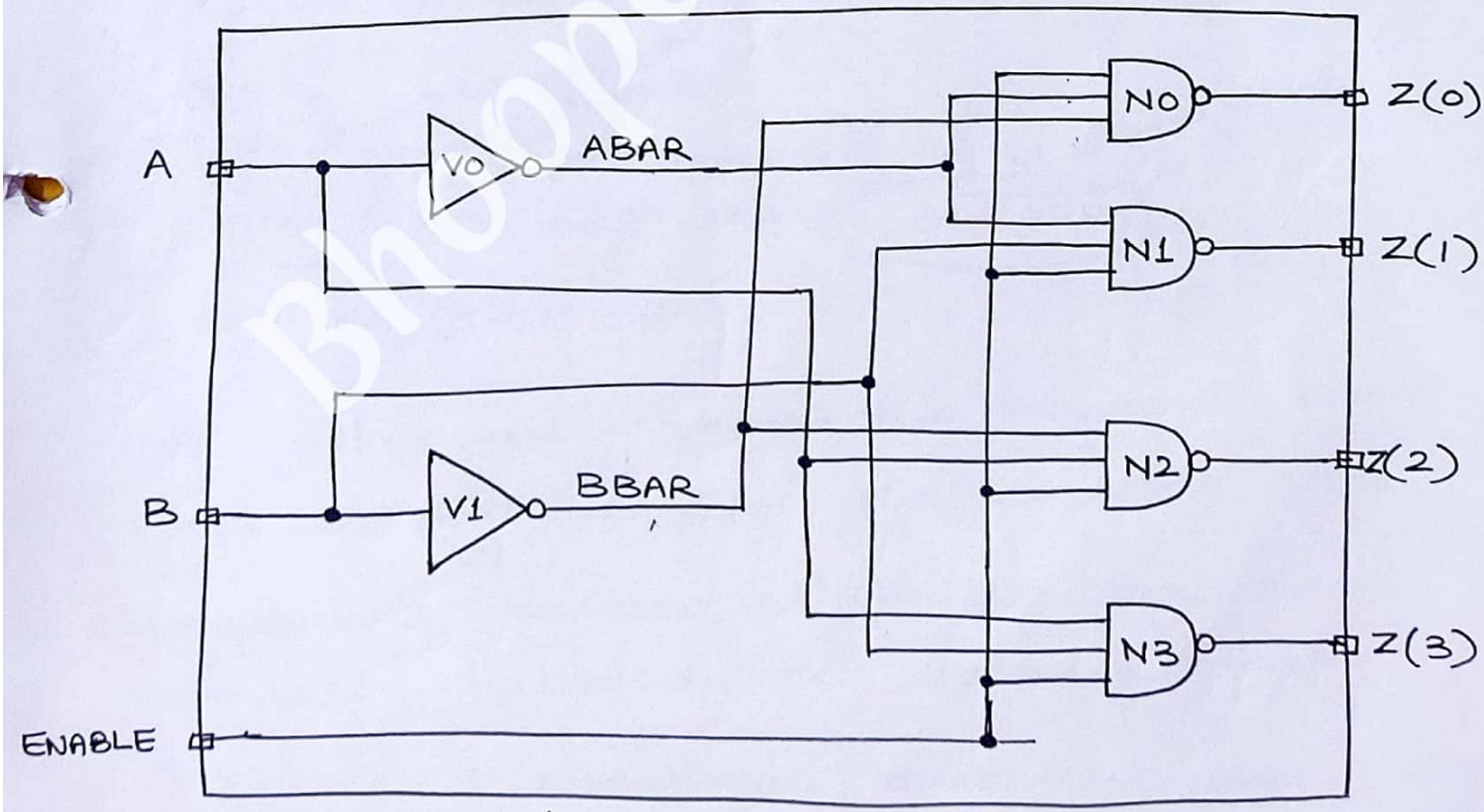
Half-Adder Circuit

```
entity HALF_ADDER is                    → name of the entity
port (A, B: in BIT; SUM, CARRY: out BIT);
end HALF_ADDER;
```

A, B: input ports of Bit-type

SUM, CARRY: output ports of Bit-type

(2.) 2X4 DECODER



2-to-4 Decoder Circuit

```
entity DECODER2X4 is
    port (A, B, ENABLE : in BIT ; Z : out BIT_VECTOR (0 to 3));
end DECODER2X4;
```

Predefined unconstrained array type of BIT

(0 to 3)) ; range 0 to 3 for port Z specifies the array size.

ARCHITECTURE BODY

(i) Structural Modeling → does not specify the functionality.

EXAMPLE: Half-Adder

```
architecture HA_STRUCTURE of HALF_ADDER is
```

Architecture name

HALF_ADDER → name of Entity

Declarative Part

```
    component XOR2
        port (X, Y : in BIT ; Z : out BIT); -- Component1
    end component;
    component AND2                          -- Component2
        port (L, M : in BIT ; N : out BIT);
    end component;
```

no. of inputs

```
begin
    X1 : XOR2 port map (A, B, SUM);
    A1 : AND2 port map (A, B, CARRY);
end HA_STRUCTURE;
```

Component labels

Signals

Statement part

Declarative part – Before the keyword begin

Statement part – After the keyword begin

* Components declared in the declarative part are instantiated in the statement part

- 1st component instantiation statement shows that signals A and B (input ports of HALF_ADDER) are connected to the X and Y input ports of XOR2 component and output port Z of XOR2 component is connected to o/p port SUM of HA entity.

EXAMPLE: DECODER2X4 ENTITY

```
architecture DEC-STR of DECODER2X4 is
    component INV
        port (PIN: in BIT; POUT: out BIT);
    end component;
    component NAND3, Three input component
        port (D0, D1, D2: in BIT; DZ: out BIT);
    end component;
    signal ABAR, BBAR: BIT;
begin
    V0: INV port map (A, ABAR);
    V1: INV port map (B, BBAR);
    N0: NAND3 port map (ENABLE, ABAR, BBAR, Z(0));
    N1: NAND3 port map (ABAR, B, ENABLE, Z(1));
    N2: NAND3 port map (A, BBAR, ENABLE, Z(2));
    N3: NAND3 port map (A, B, ENABLE, Z(3));
end DEC_STR;
```

Local signals (scope restricted to architecture body)

* The above architecture body contains a declaration statement to declare two signals: ABAR and BBAR of BIT type.
- These signals are used to connect the various components that form the decoder (just like wires).
- The scope of these signals are not visible outside the architecture body.

(ii) DATA FLOW MODELING

- Flow of data through the entity is expressed using concurrent signal assignment statements.
- Assignment of a value to a signal is done using the symbol ⇐
- Expression on RHS of the statement is computed for a value and this value is then assigned to the signal on the LHS, which is called as TARGET signal.

Imp.
A concurrent signal assignment statement is executed only when any signal (on RHS of the expression) has an event on it and an event means change in signal value.

- Delay information is included in the signal assignment statements using after clauses.
- Ordering of the statements in architecture body is not of much significance as the assignment is concurrent.
- If delay information is not included, then a default delay of 0 ns (known as Delta delay) is assumed.

EXAMPLE :

(1.) HALF_ADDER Entity

```
architecture HA_CONCURRENT of HALF_ADDER is
begin
    SUM <= A xor B after 8 ns;
    CARRY <= A and B after 4 ns;
end HA_CONCURRENT;
```

Concurrent Statements

Delay Information

* In above struct, if either signal A or B has an event at time T, the RHS expressions of both signal assignment statements are evaluated. But, signal SUM will get the newvalue only when the simulation time advances to (T+8) ns and signal CARRY will get the new value only when the simulation time advances to (T+4) ns.

(2.) DECODER2X4 Entity

```
architecture DEC_DATAFLOW of DECODER2X4 is
    signal ABAR, BBAR : BIT;
begin
    Z(3) <= not (A and B and ENABLE);      -- St 1
    Z(0) <= not (ABAR and BBAR and ENABLE); -- St 2
    BBAR <= not B;                          -- St 3
    Z(2) <= not (A and BBAR and ENABLE);    -- St 4
    ABAR <= not A;                          -- St 5
    Z(1) <= not (ABAR and B and ENABLE);    -- St 6
end DEC_DATAFLOW;
```

(iii) Behavioral Modeling

- Specifies Behavior of an entity as a set of statements that are executed sequentially in the specified order and these sequential statements are specified inside a process which specifie only the functionality of the entity (not the structure).

Example: DECODER2X4 Entity

```
architecture DEC_SEQUENTIAL of DECODER2X4 is
begin
    process (A, B, ENABLE)
        variable ABAR, BBAR : BIT ;
    begin
        ABAR := not A ;
        BBAR := not B;
        if ENABLE = '1' then
            Z(3) <= not (A and B) ;
            Z(0) <= not (ABAR and BBAR) ;
            Z(2) <= not (A and BBAR);
            Z(1) <= not (ABAR and B) ;
        else
            Z <= "1111" ;
        end if;
    end process;
end DEC_SEQUENTIAL;
```

Declarative Part

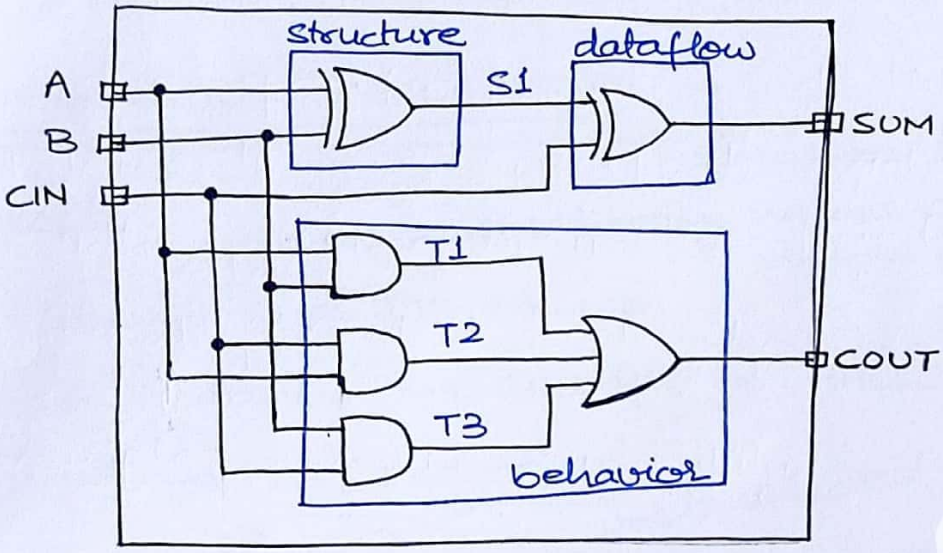
Statement Part of Process Statement

scope is also limited to that process only.

different from a signal as it is always assigned a value instantaneously and it is done with the assignment operator with := symbol.

MIXED STYLE of MODELING

Example: FULL-ADDER Entity




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

architecture FA_MIXED of FULL_ADDER is
    component XOR2
        port (P1, P2: in BIT; PZ: out BIT);
    end component;
    signal S1: BIT;
begin
    X1: XOR2 port map (A, B, S1);   <- Structural
    process (A,B,CIN)
        variable T1, T2, T3: BIT;   -> local variables or signals
    begin                                  (Sequential statements)
        T1 := A and B;
        T2 := B and CIN;
        T3 := A and CIN;
        COUT <= T1 or T2 or T3;
    end process;                           (Behavioral)
```

```
    SUM <= S1 xor CIN;
end FA_MIXED;
```

} dataflow

CONFIGURATION DECLARATION

→ Specifies Binding

Example:

```
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;
```

- library clause makes library names CMOS_LIB and MY_LIB visible within configuration declaration.
- HA_STRUCTURE is selected for configuration-binding.
- Component Binding
- X1 → label for component instantiation
- CMOS_LIB.XOR_GATE(DATAFLOW); → Entity-Architecture pair which resides in design library
- MY_LIB.AND_CONFIG; → configuration residing in design library

* first statement (for X1: ... end for) binds the component instantiation with label X1 to an entity represented by entity-architecture pair, the XOR_GATE entity declaration and the DATAFLOW architecture body, which resides in the CMOS-LIB design library.

* In second statement (for A1: AND2 ... end for), 'A1' component instantiation is bound to a configuration of an entity defined by configuration declaration, with name AND_CONFIG residing in MY_LIB design library.

PACKAGE DECLARATION

Includes common declarations e.g. types, components, procedures and functions which can further be imported to other design units using a use clause.

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 ;
```

- constant name
- value to be assigned at the start of simulation.
- TIME : type
- INTEGER : type
- function name
- return type

How to use a PACKAGE

To import all declarations:

```
library DESIGN_LIB ;
use DESIGN_LIB.EXAMPLE_PACK.all;
entity RX is ...
```

- Assuming that the above package is compiled into design library called DESIGN_LIB
- name of library

* use clause here imports all declarations in EXAMPLE_PACK into entity declaration of RX.

How to import selective declarations from a Package

Example 1:

```
library DESIGN_LIB ;
use DESIGN_LIB.EXAMPLE_PACK.D_FLIP_FLOP;
use DESIGN_LIB.EXAMPLE_PACK.PIN2PIN_DELAY;
architecture RX_STRUCTURE of RX is ...
```

Example 2:

```
library DESIGN_LIB;
package ANOTHER_PACKAGE is
    function POCKET_MONEY (MONTH: DESIGN_LIB.EXAMPLE_PACK.SUMMER)
                return INTEGER;
    constant TOTAL_ALU: INTEGER;
end ANOTHER_PACKAGE;
```

return type

type declared in EXAMPLE_PACK package is imported

*No need to use a "use" clause

This statement defines a constant but the value of the constant is not specified here; such a constant is known as 'Deferred Constant'. (value of such a constant will be specified in corresponding package body).

**ASSIGNMENT

To read about various functions, types, subtypes defined in STD_LOGIC_1164 package.

PACKAGE BODY

- Used to define the functionality and procedures declared in package declaration and also to provide value to constants declared as Deferred.
- Name of package body and package declaration must be same.
- Package body must be unique i.e. a package-declaration can have at most one package body associated with it.

BASIC LANGUAGE ELEMENTS

1.) IDENTIFIERS → Nouns (Names) used to identify data struc., objects, blocks, statements, etc. in VHDL.

(a) **Basic Identifier** :→ composed of sequence of one or more characters (Except keywords).

Rules: (i) Legal characters are: Uppercase Letters (A..Z), Lowercase letters (a...z), digits (0...9) and underscore (_) character.

(ii) 1st character must be a letter and last letter cannot be an underscore.

(iii) Two underscore characters cannot appear consecutively.

(iv) Upper-case and Lower-case characters are identical.

Examples: RAM_address, r2d2, SET_Ck_High

(b) **Extended Identifier** :→ Sequence of characters written between two backslashes (\).

Rules: (i) All characters are legal including keywords.

(ii) within two backslashes (\), upper-case and lower-case letters are considered distinct.

(iii) Extended identifiers and basic identifiers are different if defined with same name.

(iv) Two consecutive backslashes represents one backslash.

Examples: \2for$\ , \process\ , \---\\---\

DATA OBJECTS

→ Used to hold a value of a specified type.

Classes of Data Objects :→

(i) Constant : An object with 'constant' class can hold a single value of a given type (Assignment is before simulation)

(ii) Variable : An object of 'variable' class can also hold a single value of a given type. But different values can be assigned at different times using variable-assignment statement.

(iii) Signal : An object of 'signal' class can hold a list of values (current and possible future values that are likely to appear on a signal)

(iv) file : A "file" object contains a sequence of values that can be read or written to some file.

Declaration Methods for Data objects

(i) for Constants → with keyword "constant"

Examples:

Name of Constant

```
constant RISE_TIME : TIME := 10 ns ;
constant BUS_WIDTH : INTEGER := 8 ;
```

} Specified value is assigned to object at time of simulation (start)

```
* constant NO_of_Inputs : INTEGER ;
```

Type (predefined in VHDL)

→ Value of the constant is not specified for assignment and such a constant is known as "Deferred constant".

* ↳ can appear only in package declaration.

(ii) for Variables → with keyword "variable"

Examples

```
variable CTRL_STATUS: BIT_VECTOR (10 downto 0);
```

CTRL_STATUS: Variable Name; BIT_VECTOR (10 downto 0): specifies array of 11 elements of type BIT.

```
Variable SUM: INTEGER range 0 to 100 := 10;
```

SUM: V-Name; INTEGER: Type; range 0 to 100: Possible range of values; := 10: initial value assignment (if not specified then default value T'LEFT is assigned)

where, T → Object Type and LEFT → predefined leftmost value from the set of T-type object.

```
Variable FOUND, DONE: BOOLEAN;
```

FOUND, DONE: Two variables of type Boolean will take default value at start of simulation.

* for array-type, default value is '0'
for Boolean-type, default value is 'FALSE'.

(iii) for Signals → with keyword "signal"

Examples

```
signal CLOCK: BIT;
signal DATA_BUS: BIT_VECTOR (0 to 7);
signal GATE_DELAY: TIME := 10 ns;
signal INIT_P: STD_LOGIC_VECTOR (7 downto 0) := (0 => '1',
                                                 others => 'U');
```

others => 'U': 0th element is initialized to '1' and others are initialized to value 'U'.

(iv) for files → with "file" keyword

Syntax:

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

- file → Keyword
- open → Keyword
- mode → Specifies read-only or write-only as READ or WRITE respectively or APPEND for append-mode.
- string-expression → ;

Examples

Defines file type Declaration

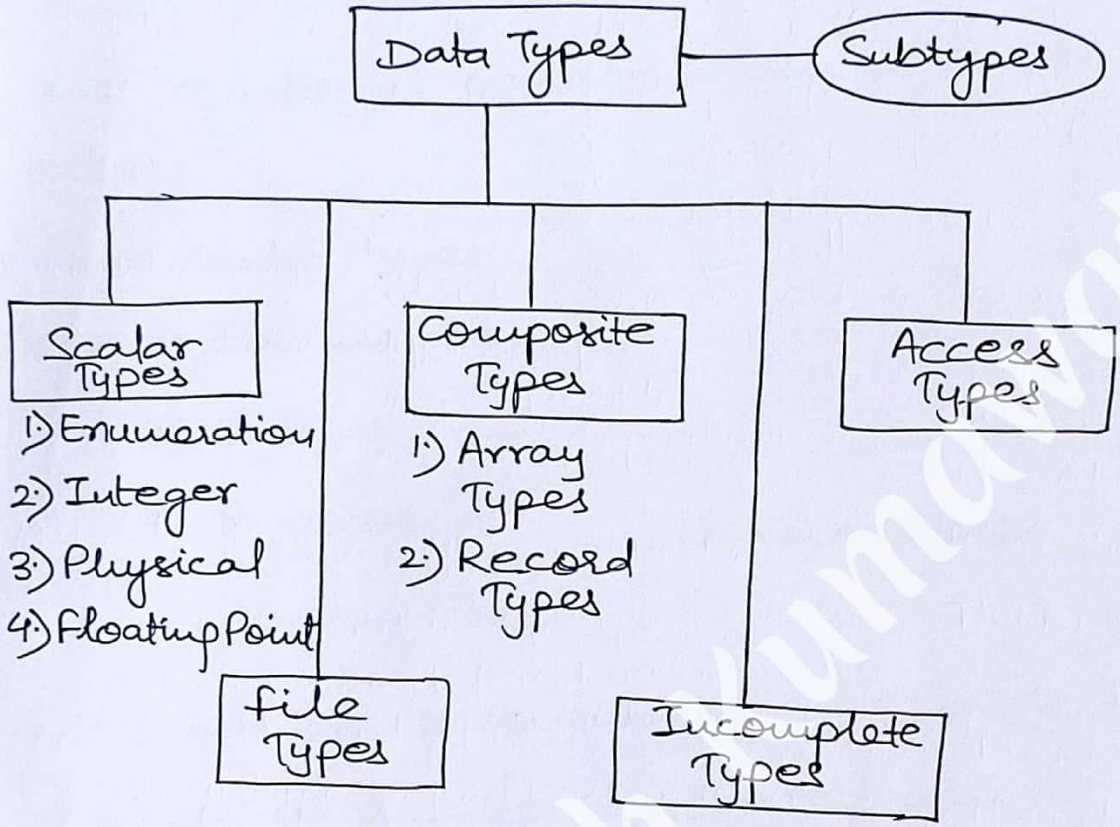
```
type STD_LOGIC_FILE is file of STD_LOGIC_VECTOR;
type BIT_FILE is file of BIT_VECTOR;
```

file Declarations

```
file STIMULUS: TEXT open READ_MODE is "/usr/home/jb/add.sti";
```

- STIMULUS → name of file
- open → Keyword
- READ_MODE → mode of opening
- "/usr/home/jb/add.sti" → Path name to a physical file in the host environment

DATA TYPES



SubType :→ A type with constraint that specifies the subset of values for the subtype where the type is called the "**Base type**" of the subtype.

Example

subtype MY_INTEGER is INTEGER range 48 to 56;
(subtype of base-type; range constraint)

type DIGIT is ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');

subtype MIDDLE is DIGIT range '3' to '7';
(user-defined enumeration type)

Subtype with basetype 'DIGIT' with values '3', '4', '5', '6' and '7'.

↳ consists of a set of user-defined values consisting of identifiers and character literals.

SCALAR TYPES

→ values belonging to 'scalar' types are ordered and relational operators can be used on these values.

→ four scalar types

Discrete Types
1.) Enumeration
2.) Integer

Numeric Types
3.) Physical
4.) Floating Point

→ Every value belonging to ①, ② or ③ type has a position number associated with it.

1.) ENUMERATION TYPES

→ A set of user-defined values consisting of identifiers and character literals.

Position Numbers → 0 1 2 3 4 5

```
type MICRO_OP is (LOAD, STORE, ADD, SUB, MUL, DIV);
subtype ARITH_OP is MICRO_OP range ADD to DIV;
```

MICRO_OP → Base Type

Object declaration

```
type MVL is ('U','0','1','Z');
signal CONTROL_A: MVL;
signal CLOCK: MVL range '0' to '1';
```

Implicit Subtype declaration (within Object declaration)

'clock' signal is of type 'MVL' that can take values in the range '0' to '1'

NOTE: In 'MICRO_OP' type declaration, we can conclude that

STORE < DIV is True
But SUB > MUL is False

} Due to Positional value association to value.

The position number of the leftmost element is 0 and it increases towards right.

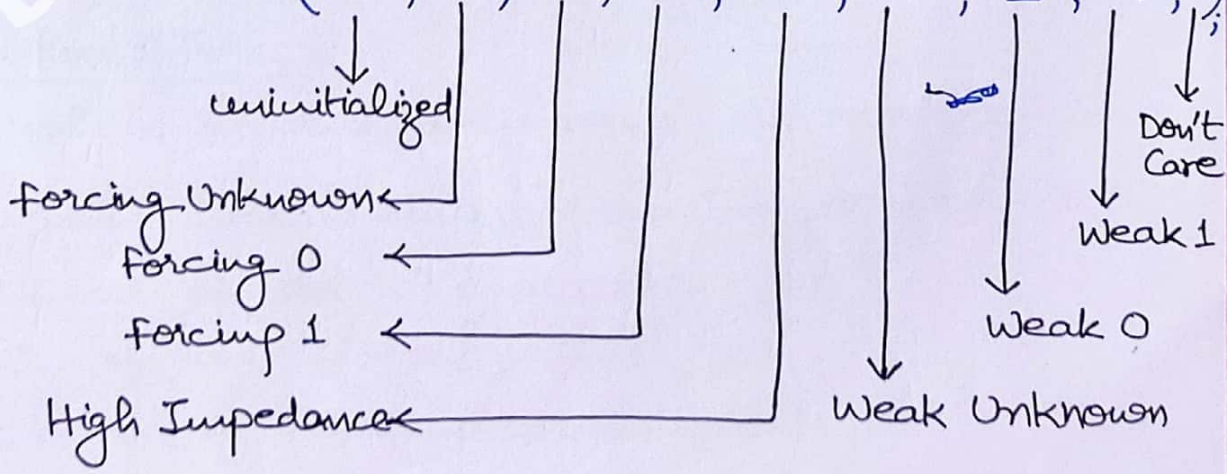
Predefined VHDL Enumeration Types

- CHARACTER → 191 eight-bit coded character-set known as character literals and are written within quotes(' ')
- BIT → Literals '0' and '1'
- BOOLEAN → Literals FALSE and TRUE
- SEVERITY_LEVEL → Literals NOTE, WARNING, ERROR and FAILURE. (used in* assertion statements)
- FILE_OPEN_KIND → READ_MODE, WRITE_MODE and APPEND_MODE (for file operations)
- FILE_OPEN_STATUS → OPEN_OK, STATUS_ERROR NAME_ERROR and MODE_ERROR

Examples :→ 'A', '_', '1', '3' characters Literals

NOTE: Type STD_ULOGIC defined in STD_LOGIC_1164 package is an Enumeration type defined as:

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



2) Integer Types

→ Set of values within the integer range from $-(2^{31}-1)$ to $+(2^{31}-1)$.

→ Values belonging to an integer type are called **integer literals**

NOTE: Predefined Integer type of VHDL is INTEGER

Examples:→ 56349

6E2 Decimal value $6*(10^2) = 600$

0

98_71_28 same as 987128

Integer Type Declarations

```
type INDEX is range 0 to 15;
type WORD_LENGTH is range 31 downto 0;
subtype DATA_WORD is WORD_LENGTH range 15 downto 0;
type MY_WORD is range 4 to 6;
```

* value 31 is at position 31 and 14 is at position 14 and so on.

* Position numbers of values 4, 5 and 6 are 4, 5, and 6 respectively.

* Difference between 'to' and 'downto'

Object Declarations

```
constant MUX_ADDRESS: INDEX := 5;
signal DATA_BUS: DATA_WORD;
```

(Type → INDEX)

3) FLOATING POINT TYPES

→ Set of values in the given range of real numbers:

→ Floating point literals differ from Integer literals by the presence of dot (.) character.

e.g. 0 is an Integer, But

0.0 is a floating-point literal.

NOTE:→ Only predefined floating point type in VHDL is REAL

Range of REAL -1.0E38 to +1.0E38

Examples of floating point Declarations

```
type TTL_VOLTAGE is range -5.5 to -1.4;
type REAL_DATA is range 0.0 to 31.9;
subtype RDIG is REAL_DATA range 0.0 to 15.9;
...
variable LENGTH: RDIG;
...
variable L1, L2, L3: RDIG;
```

Examples of floating-point Literals

15.59

0.0 → Different from 0

0.4327

2_5.5_9 → 25.59

62.7E-4 → 62.7×10^{-4}

4.0E+2 → $4.0 \times 10^{2} = 104.0$

BASED LITERALS

- Integers and floating point literals can also be written in a base other than 10 (decimal).
- The base can have any value between 2 and 16. Such literals are called as "Based Literals".
- The exponent in this case now represent a power of the specified Base.
- SYNTAX

base # based-value # form 1

Or

base # based-value # E exponent form 2

Examples

2#101_101_000# → $(101101000)_2 = (360)_{10}$

16#FA# → $(FA)_{16} = (11111010)_2 = (250)_{10}$

16#E#E1 → $(E)_{16} * (16)^1 = 14 * 16 = (224)_{10}$

2#110.01# → $(110.01)_2 = (6.25)_{10}$

4) PHYSICAL TYPES

- Values that represent measurement of some physical quantity like time, length, voltage or current.
- Values of this type are expressed as integer multiples of a base unit.
- Values of a physical type are called as physical literals
- Only predefined physical type is TIME, and its range is between $-(2^{31}-1)$ to $+(2^{31}-1)$
- A predefined physical sub-type is also there named as DELAY_LENGTH which represents non-negative time values

Examples

(1.)
```
type CURRENT is range 0 to 1E9
    units
        nA;                    ← Base Unit
        uA = 1000 nA;          ← Derived Units
        mA = 1000 uA;
        Amp = 1000 mA;
    end units;
```

(2.)
```
type STEP_TYPE is range -10 to +10
    units
        STEP;
        STEP2 = 2 STEP;
        STEP5 = 5 STEP;
    end units;
```

```
subtype FILTER_CURRENT is CURRENT range 10 uA to 5 mA;
```

* Here, nA represents 1 nA and STEP represents 1 step.

COMPOSITE TYPES

- Represents a collection of values
- Two composite types are: **Array Type**: all values belonging to a single type

  Record Type: Values may belong to different type

(i) ARRAY TYPES

Examples :→

type ADDRESS_WORD is array (0 to 63) of BIT;

(Keyword: is array; range: 0 to 63; Type of array elements: BIT)

type DATA_WORD is array (7 downto 0) of STD_ULOGIC;

type ROM is array (0 to 125) of DATA_WORD;

Object Declarations

variable ROM_ADDR : ROM;

signal ADDRESS_BUS : ADDRESS_WORD;

constant DECODER : DECODE_MATRIX; → Deferred Constant

→ ADDRESS_BUS is a one-dimensional array object that consists of 64 elements of BIT type.

→ ROM_ADDR is a one-dimensional array object that consists of 126 elements with each element being another array object consisting of 8 elements of type STD_ULOGIC.

Accessing an Array Element

- Array element can be accessed by specifying the index value into the array.

Example :→ ADDRESS_BUS(26) refers to the 27th element of ADDRESS_BUS array object.

ROM_ADDR(10)(5) refers to the value at index 5 of the ROM_ADDR(10) data object i.e. index 5 value at 11th address.

Value Assignment to an Array Object

```
ROM_ADDR(5) := "01000100" ;   → An Element of array is assigned a value.
DECODE_VALUE := DECODER ;     → Entire Array is assigned
ADDRESS_BUS (8 to 15) <= X"FF" ;  → A slice(portion) of an array is assigned a value.
```

UNCONSTRAINED ARRAY → * no. of elements in type, is not specified.

* Unconstrained array declaration is done through "<>" symbol called a "box".

NOTE: VHDL does not allow a type that is an unconstrained array of an unconstrained array.

Example

Invalid in VHDL

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

(NATURAL range <>) → unconstrained array

STD_LOGIC_VECTOR; → unconstrained array type

One-Dimensional Unconstrained Array types defined in package STD_LOGIC_1164 are:

1. type STD_ULOGIC_VECTOR is array (NATURAL range<>) of STD_ULOGIC;

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

Predefined One-Dimensional Unconstrained Array types defined in VHDL are: STRING and BIT_VECTOR.

Example :→

variable MESSAGE : STRING (1 to 17) := "Hello, VHDL World"; ← String Literal

String Literal :→ A value representing a 1-D array of characters and these are written by enclosing the sequence of characters within double quotes ("...").

RECORD TYPES (same as struct in C)

- composed of elements of same or different types

Example :→

```
type PIN_TYPE is range 0 to 10;
type MODULE is record  SIZE : INTEGER range 20 to 200;
                       CRITICAL_DELAY : TIME;
                       NO_INPUTS : PIN_TYPE;
                       NO_OUTPUTS : PIN_TYPE;
end record;
```

Value Assignment to Record object

→ Done using Aggregates.

AGGREGATE

An Aggregate is a set of comma-separated elements enclosed within parenthesis. It can also be used to initialize an array object in its declaration.

Examples

(i) Array Assignment

variable OP_CODES : BIT_VECTOR (1 to 5);

OPCODES := "01001"; → Array is assigned to a string literal

OPCODES := ('0', '1', '0', '0', '1'); → Positional Association

Assigned to op_codes(1) ← '0', '1' → to OP_codes(2), '0' → to op_codes(3) and so on.

OP_CODES := (2 => '1', 5 => '1', others => '0'); → Named Association

(2nd ← Element Number, 5th)

OP_CODES := (others => '0'); → All values are assigned to value '0'.

constant CLN : STD_LOGIC_VECTOR (2 downto 0) := "001";

Constant object] CLN(2) → 0
CLN(1) → 0
CLN(0) → 1

(ii) RECORD ASSIGNMENT

object of record type

variable NAND_COMP : MODULE;

NAND_COMP := (50, 20 ns, 3, 2);

Assignment To → Size, Critical_DLY, NO_INPUTS, NO_OUTPUTS

BEHAVIORAL MODELING

→ Irrespective of the modeling style used, every entity is represented using an Entity-Declaration and at least one Architecture Body

ENTITY DECLARATION

→ Specifies name of entity, names of interface ports, their mode (i.e. direction) and the type of ports.

SYNTAX :→

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

Example :→

```
entity AOI is                    → AND-OR-Invert
    port (A, B, C, D: in BIT; Z: out BIT);
end AOI;
```

ARCHITECTURE BODY → Describes internal view of entity.

SYNTAX :→

```
architecture architecture-name of entity-name is
    [architecture-item-declarations]
begin
    concurrent-statements;
    :
end [architecture] [architecture-name];
```

Concurrent-Statements Includes :

process-statement
block-statement
concurrent-procedure-call-statement
concurrent-assertion-statement
concurrent-signal-assignment-statement
component-instantiation-statement
generate-statement.

PROCESS STATEMENT

→ contains sequential statements that describe the functionality of an entity in sequential terms.

Syntax:→

[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
report-statement
procedure-call-statement
return-statement
end process [process-label];

→ optional

→ set of signals to which the process is sensitive.

(i) Variable Assignment Statement

Syntax:→ variable-object := expression ;

(ii) Signal Assignment Statement

Syntax:→ signal-object <= expression [after delay-value] ;

(iii) Wait Statement

Syntax:→
- wait on sensitivity-list ;
- wait until boolean-expression ;
- wait for time-expression ;

Combination of these can also exist with syntax:→

wait on sensitivity-list until boolean-expression for time-expression ;

Examples:

1. wait on A, B, C ; → wait for occurance of an event on A, B or C
2. wait until A = B ;
3. wait for 10 ns ;
4. wait on CLOCK for 20 ns ;
5. wait until SUM > 100 for 50 ms ;
6. wait on CLOCK until SUM > 100 ;

2. → Process is suspended until the condition becomes TRUE. On occurance of an event on signal A or B, the condition is evaluated and if it is TRUE, the process resumes execution from the next statement onwards, otherwise it suspends again.

→ If wait last statement then return first statement

4. → Process is suspended and wait for an event to occur on signal CLOCK for a time-out interval of 20 ns. If no event occurs within 20 ns, the process resumes with the statement following the wait.

NOTE: wait for 0
means to wait for one Delta cycle.

(iv) If Statement

Syntax:→ if boolean-expression then
sequential-statements
{ elsif boolean-expression then
sequential-statements }
[else
sequential-statements]
end if ;

* If-statement can have 0 or more elsif clauses.

And Else clause is optional.

(v) Case Statement

Syntax:→ case expression is
can have Any number of branches { when choices => sequential-statements
when choices => sequential-statements
:
[when others => sequential-statements] ↳ Last Branch
end case ;

Example:→

```
type WEEK_DAY is (MON,TUE,WED,THU,FRI,SAT,SUN);
type DOLLARS is range 0 to 10;
variable DAY: WEEK_DAY;
variable POCKET_MONEY: DOLLARS;
case DAY is
   when TUE          => POCKET_MONEY := 6;
   when MON | WED    => POCKET_MONEY := 2;
   when FRI to SUN   => POCKET_MONEY := 7;
   when others       => POCKET_MONEY := 0;
end case;
```

) NULL Statement

null ; → This sequential statement does not cause any action to take place. It may be specified where it is necessary to explicitly specify that no action needs to be performed.

(vii) Loop Statement

Syntax:→ [loop-label] iteration-scheme loop
sequential-statements
end loop [loop-label] ;

Iteration-Schemes

(a.) for iteration scheme

syntax:→

for identifier in range

Example

```
FACTORIAL := 1 ;          Identifier (* no explicit declaration is necessary and scope is also within this loop)
for NUMBER in 2 to N loop
                  range
    FACTORIAL := FACTORIAL * NUMBER ;   ] Body of the loop
end loop ;
```

→ In above example, the body of the for loop is executed (N-1) times with NUMBER being incremented by 1 at the end of each iteration.

(b.) while iteration scheme

Syntax :→ while boolean-expression

Example

```
J:=0; SUM:=10;
WH-LOOP: while J<20 loop
    SUM := SUM * 2;
    J := J+3;
end loop;
```

Label of loop → WH-LOOP

] Body of the loop

→ Here, the statements within the body of the loop are executed sequentially and repeatedly till the loop-condition, J<20, is TRUE. when loop condition becomes FALSE, execution continues with the statement following the loop statement.

(c.) Simple loop and Exit iteration scheme

↳ all statement execute repeatedly until some other action calls the loop to exit

Example

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

loop label → L2

] Body of loop

→ Next statement / Return Statement / Exit statement causes the execution to jump out of loop according to condition specified.

* In absence of an exit-statement, the loop would execute indefinitely.

Exit statement Sequential → used inside the loop.

syntax:→ exit [loop-label] [when condition]; (jump outside the loop or at specified label)

* If no loop-label is specified, the innermost loop is exited according to the condition specified through when clause, if it is present. (if condition true)

Example

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

(label)

(if condition true else continue to next statement)

Next Statement → used to skip the remaining statements in current iteration.

syntax:→ 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;
    K := K+1;
end loop;
```

In the example, when the next statement is executed, the statements after the next statement will not be executed and it come out of the loop without executing K := K+1 .

The value of J (loop identifier) now is decremented and the loop again will be executed with new value of J.

VHDL code for logic Gates

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

entity and_gate is
    port (a: in STD_LOGIC;
          b: in STD_LOGIC;
          c: out STD_LOGIC);
end and_gate

architectural Behavioral of and_gate is
begin
    process (a, b)
    begin
        c <= a and b;
    end process;
end behavioral;
```

For OR gate
`c <= a or b;`
For NOR gate
`c <= a nor b;`
For NAND gate
`c <= a nand b;`
For XOR gate
`c <= a XOR b;`
For X-NOR gate
`c <= a X-NOR b;`

For NOT gate

```
entity Not_gate is
  port (a: in STD_LOGIC;
        b: out STD_LOGIC);
end Not_gate
architectural Behavioral of Not_gate is
begin
    process (a)
    begin
        b <= not a;
    end process;
end behavioral
```

HALF-ADDER

```
library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.std_logic_Arith.ALL;
use IEEE.std_logic_unsigned.ALL;

entity Half_Adder is
  port(a: in STD_LOGIC
       b: in STD_LOGIC
       SUM: out STD_LOGIC
       CARRY: out STD_LOGIC);
end Half_Adder

architectural Behavioral of Half_Adder is
begin
  process
  begin
  sum <= a xor b;
  carry <= a and b;
  end process;
end behavioral;
```

full-Adder

```
entity full_Adder is
  port(a: in STD_LOGIC
       b: in STD_LOGIC
       c: in STD_LOGIC
       SUM: out STD_LOGIC
       CARRY: out STD_LOGIC);
end full_Adder;
architectural Behavioral of full_Adder is
begin
    process(a,b,c)
    begin
    sum <= a xor b xor c;
    CARRY <= (a and b) or (b and c) or (c and a);
```

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

Using Intermediate Code

```
architectural Behavioral of Full_Adder is
begin
process (a,b,c)
variable x,y,z : STD_LOGIC;
begin
    sum <= a XOR b XOR c;
    x := a and b;
    y := b and c;
    z := a and c;
    carry <= x OR y OR z;
    end process;
    end Behavioral;
```

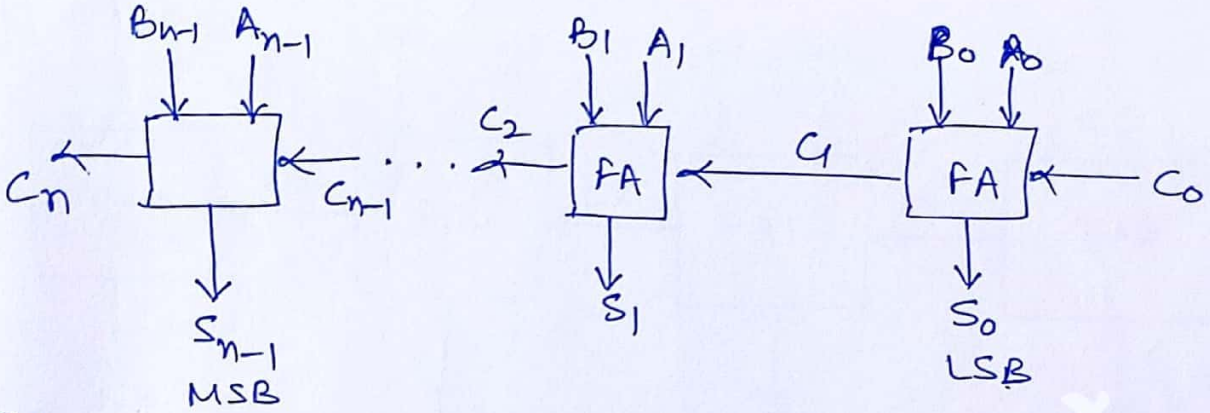
Full Adder using Intermediate Code

```
architecture Behavioral of FULL_ADDER is
begin
process (a, b, c)
variable  x, y, z : STD_LOGIC;
 begin
      SUM <= a XOR b XOR c;
      x := a and b;
      y := b and c;
      z := a and c;
      carry <= x or y or z;
  end process;
  end behavioral;
```

full Subtractor using Intermediate Code

```
architecture Behavioral of FULL_SUBTRACTOR is
begin
process (a, b, c)
variable  x, y, z : STD_LOGIC;
begin
     DIFF <= a  XOR b  XOR c;
      x := (NOT a) and b;
      y :=  b and c;
      z :=  c and (not a);
    BORROW <= x or y or z;
  end process;
end Behavioral;
```

Ripple Adder



FA

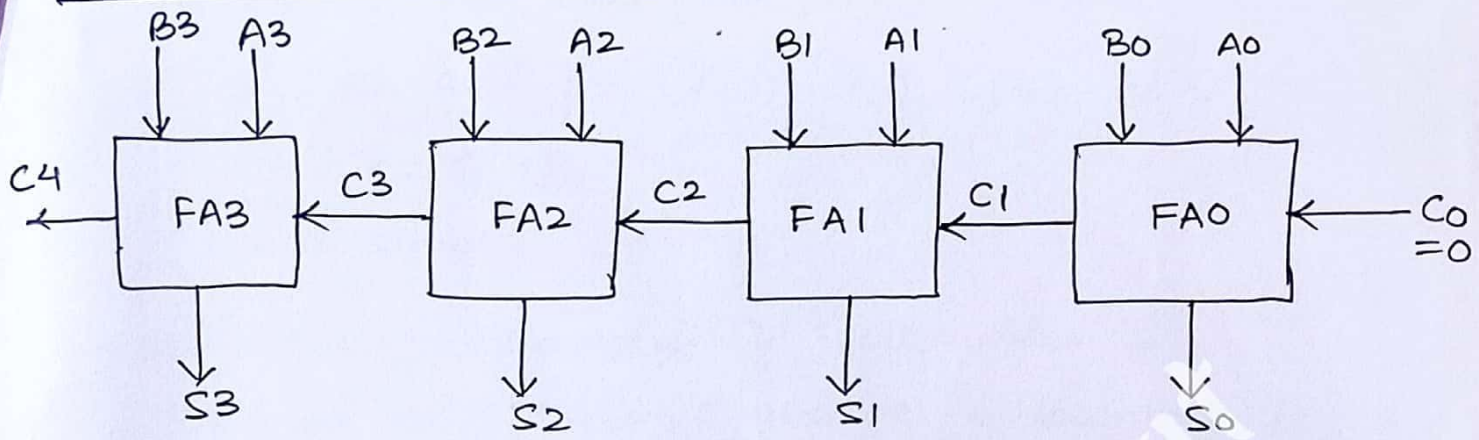
$$Sum = A_i \oplus B_i \oplus C_i$$

$$C_{out} = C_{i+1} = A_i B_i + (A_i \oplus B_i) C_i$$

```
Entity R_ADDER is                                   for 8-bit RippleAdder
    Port ( A,B : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
           Cin : In STD_LOGIC;
           Sum : out STD_LOGIC_VECTOR (7 Downto 0);
           Cout : OUT STD_LOGIC );
end R_ADDER;

Architecture Behavioral of R_ADDER is
    signal carry: STD_LOGIC_VECTOR (8 down to 0);
    Begin
        carry(0) <= Cin ;
        Process (A, B, Cin)
        for i IN 0 to 7 Loop
        Sum(i) <= A(i) XOR B(i) XOR carry(i) ;
        carry(i+1) <= (A(i) and B(i)) OR (carry(i) and
                                          (A(i) XOR B(i)) ;
        END LOOP;
        END PROCESS;
    END Behavioral;
```

4-BIT RIPPLE CARRY ADDER



full-Adder

$$Sum = A_i \oplus B_i \oplus C_i$$

$$C_{out} = C_{i+1} = A_i B_i + (A_i \oplus B_i) C_i$$

VHDL CODE

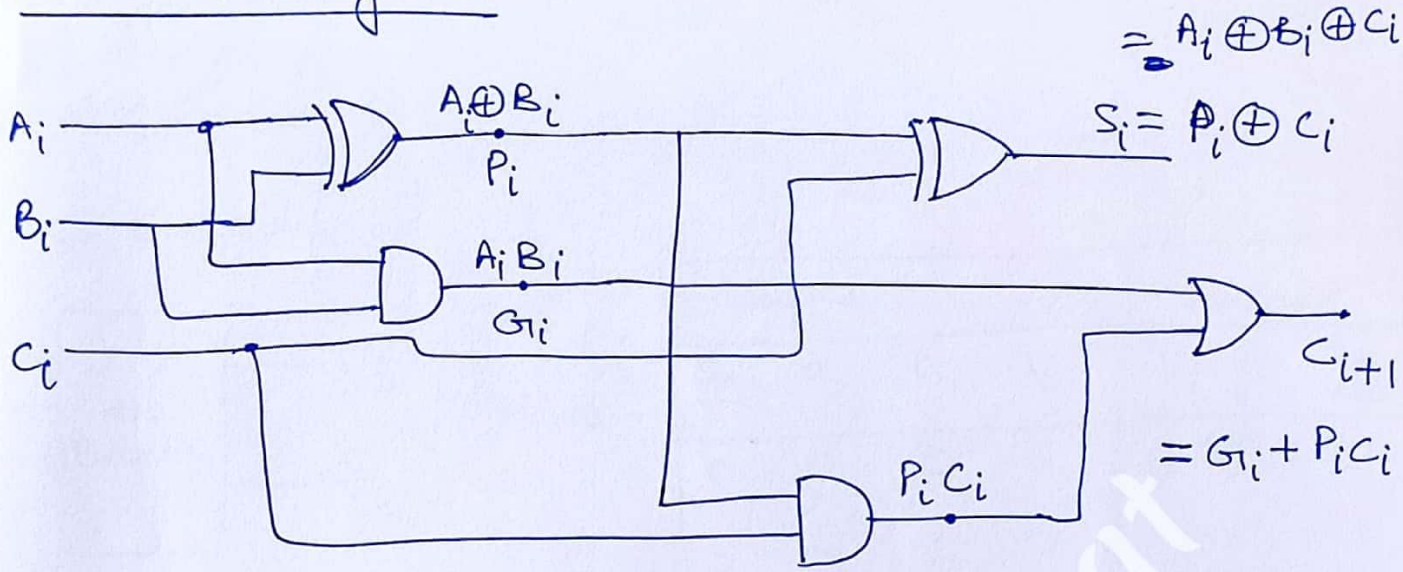
```
entity 4bit_radder is
    port ( A,B : IN STD_LOGIC_VECTOR (3 downto 0);
           Cin : IN STD_LOGIC;
           SUM : OUT STD_LOGIC_VECTOR (3 downto 0);
           Cout : OUT STD_LOGIC);
end 4bit_radder;

architecture struct of 4bit_radder is
    signal c : STD_LOGIC_VECTOR (4 downto 0);
    Component FA is
      port (A, B, Cin : IN STD_LOGIC;
            SUM, COUT : OUT STD_LOGIC;
      end component;
    begin
      C(0) <= '0';
      F1 : FA port map (A(0), B(0), C(0), S(0), C(1));
      F2 : FA port map (A(1), B(1), C(1), S(1), C(2));
```

```
    F3 : FA port map ( A(2), B(2), C(2), S(2), C(3));
    F4: FA port map ( A(3), B(3), C(3), S(3), C(4));
    Cout <= C(4);
    end struct;

architecture Behavioral of 4bit_adder is
    signal C : STD_LOGIC_VECTOR (4 downto 0);
    Begin
    C(0) <= Cin;
    process (A, B, Cin)
    C(0) <= '0';
    for i IN 0 to 3 loop
        SUM(i) <= A(i) XOR B(i) XOR C(i);
        CARRY(i+1) <= ( A(i) and B(i)) OR ( C(i) and
                                    (A(i) XOR B(i));
    end loop;
    end process;
end Behavioral;
```

Look Ahead Carry Adder




```
Entity LAC_ADDER is
    Port( a,b: IN STD_LOGIC_VECTOR(7 downto 0);
          Cin: IN STD_LOGIC;
          Sum: OUT STD_LOGIC_VECTOR(7 downto 0);
          Cout: OUT STD_LOGIC);
End LAC_ADDER;
Architecture Behavioral of LAC_ADDER is
    signal generator: STD_LOGIC_VECTOR(7 downto 0);
    signal propagation: STD_LOGIC_VECTOR(7 downto 0);
    signal Carry: STD_LOGIC_VECTOR(8 down to 0);
Begin
    Process(A,B,Cin)
    Begin
    generator <= A(i) and B(i);
    Propagation <= A(i) xor B(i);
    Carry(0) <= Cin;
    for i In 1 to 8 LOOP
    Process(
    Carry(i) <= Generator(i-1) OR (Propagation(i-1) and
                Carry(i-1));
    End LOOP;
    End Process;
    SUM <= Propagation(
```

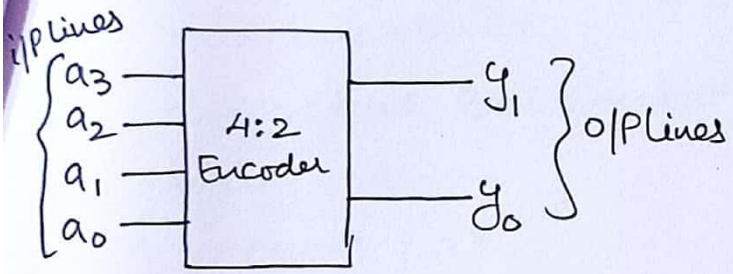
Encoder

a.) 4:2 Encoder

b.) 8:3 Encoder

c.) 16:4 Encoder

(a.) 4:2 Encoder



Input				Output	
a_3	a_2	a_1	a_0	y_1	y_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

VHDL Code :→

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

Entity encoder4x2 is
   port (a: in std_logic_vector (3 downto 0);
         y: out std_logic_vector (1 downto 0));
end encoder4x2;
Architecture Behavioral of encoder4x2 is
begin
   process (a)
   begin
      case a is
      when "0000" => y <= "00";
      when "0010" => y <= "01";
      when "0100" => y <= "10";
      when "1000" => y <= "11";
   end case;      end process; end Behavioral;
```

8:3 Encoder

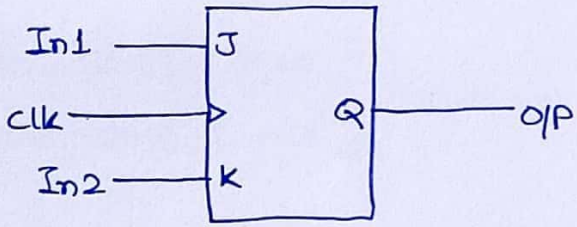
```
Entity encoder8x3 is
  port (a: in std_logic_vector (7 downto 0);
        y: out std_logic_vector (2 downto 0));
  end encoder8x3;
Architecture Behavioral of encoder8x3 is
begin
  process (a)
  begin
    case a is
      when "00000001" => y <= "000";
      when "00000010" => y <= "001";
      when "00000100" => y <= "010";
      when "00001000" => y <= "011";
      when "00010000" => y <= "100";
      when "00100000" => y <= "101";
      when "01000000" => y <= "110";
      when "10000000" => y <= "111";
    end case;
  end process;
end Behavioral;
```

4x1 MUX

```
Entity MUX is
  port (X0,X1,X2,X3 : in STD_LOGIC;
        S: in STD_LOGIC_VECTOR(1 downto 0);
        f: out STD_LOGIC);
end MUX;
Architecture Behavioral of MUX is
begin
  Process (S, X0, X1, X2, X3)
  begin
     case S is
        when "00" => f <= X0;
        when "01" => f <= X1;
        when "10" => f <= X2;
        when "11" => f <= X3;
     end case;
  end process;
end Behavioral;
```

VHDL codes

1.) JK Flip-flop




```
entity JK_ff is
port (J, K, clk : in BIT;
      Q : out BIT);
end JK_ff;
Architecture Behavioral of JK_FF is
begin
  process (J, K, CLK)
  begin
    If (clk 'event and clk = '1') then
      If (J=0 and K=0) then
        Q <= Q;
      Elsif (J=1 and K=0) then
        Q <= 1;
      Elsif (J=0 and K=1) then
        Q <= 0;
      elsif (J=1 and K=1) then
        Q <= not Q;
      end if;
    end if;
  end process;
end Behavioral;
```

```
entity SR_ff is
port (S, R, clk : in std_logic;
      Q, QBAR: out std_logic);
end SR_ff;

Architecture behavioral of SR_ff is
begin
  process (clock)
  variable tmp: std_logic;
  begin
   if (clock='1' and clock'Event') then
      if (S='0' and R='0') then
       tmp := tmp;
       elsif (S='0' and R='1') then
       tmp := 0;
       elsif (S='1' and R='1') then
       tmp := 'Z';
       else
       tmp := '1';
       end if;
    end if;
   Q <= tmp;
   QBAR <= not tmp;
  end process;
end behavioral;
```

Date .....

Expt. No. ..... Page No. .....

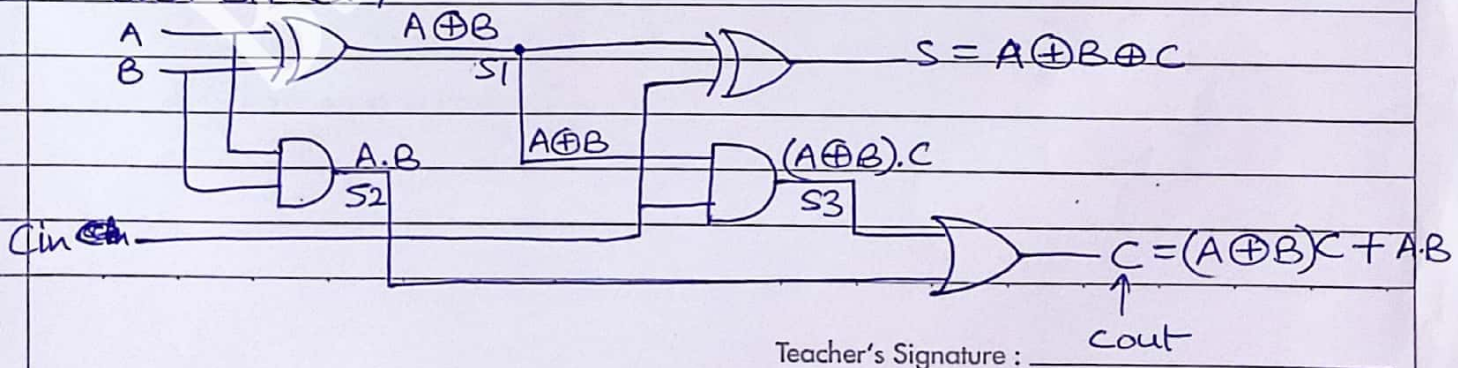
Full-Adder → (ii) Structural

```
architecture struct of fa is
component 1 { component and21
                port (a,b : in std_logic ; c : out std_logic);
              end component ;

component 2 { component XOR21
                port (a,b : in std_logic ; c : out std_logic);
              end component ;

component 3 { component OR31
                port (a, b, c : in std_logic ; d : out std_logic);
              end component ;

signal S1, S2, S3 : std_logic ;
begin
U1 : XOR21 port map (a, b, S1) ;
U2 : XOR21 port map (S1, Cin, S) ;
U3 : and21 port map (a, b, S2) ;
U4 : and21 port map (S1, Cin, S3) ;
U5 : OR31 port map (S2, S3, Cout) ;
end struct ;
```



Teacher's Signature : _____

Expt. No..... Page No.....

Half-Adder

```
entity ha is
  port (a, b : in STD_LOGIC ; S, Cout : out STD_LOGIC);
end ha;
architecture behavioral of ha is
 begin
      S <= a xor b;
      Cout <= a and b;
 end behavioral;
```

Full-Adder → (i) behavioral

```
entity fa is
  port(a,b,cin: in STD_LOGIC ; S,Cout : out STD_LOGIC);
end fa;

architecture behavioral of fa is
 begin
    process (a, b, cin)
    begin
       S <= a xor b xor c;
       Cout <= (a and b) or (b and cin) or
                                (cin and a);
    end process;
end behavioral;
```

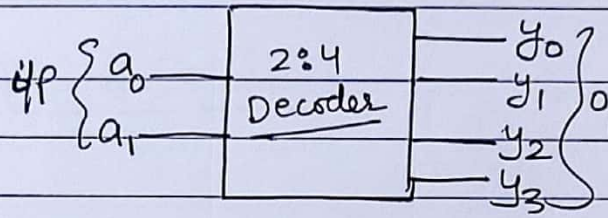
Teacher's Signature : _____

Date _____

Expt. No. _____ Page No. _____

DECODERS

1) 2:4 Decoder




```
Entity decoder2x4 is
  port ( a: in std_logic_vector (1 downto 0) ;
         y: out std_logic_vector (3 downto 0));
end decoder2x4 ;

architecture Behavioral of decoder2x4 is
 begin
    process (a)
    begin
       case a is
          when "00" => y <= "1000" ;
          when "01" => y <= "0100" ;
          when "10" => y <= "0010" ;
          when others => y<= "0001" ;
       end case;
    end process;
 end behavioral;
```

2) 3:8 decoder

Entity declaration:
```
a: in std_logic_vector (2 downto 0);
y: out std_logic_vector (7 downto 0);
```

Process:
```
case a is    { 8 Cases ranging from "000" to "111"
y ranging from "10000000"
               "01000000"
               :
               :
               "00000001"
```

Teacher's Signature : _____

Date _____

Expt. No. _____ Page No. _____

with the use of if-else

```
process (a)
   begin
      if (a = "000") then y <= "10000000";
   elsif (a = "001") then y <= "01000000";
   elsif  :
   elsif  :
   elsif  :
          :
          :
   else y <= "00000001";
   end if;
end process;
```

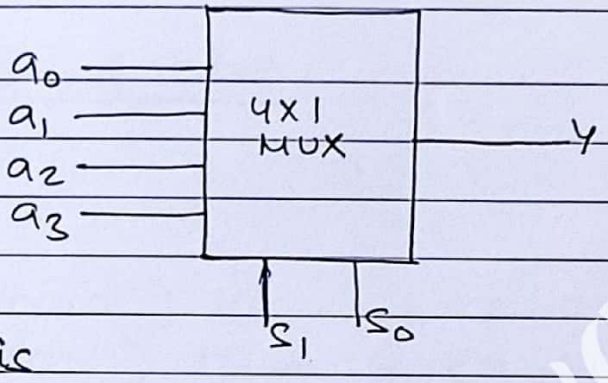

Teacher's Signature : _____

Date _____

Expt. No. _____ Page No. _____

Multiplexers

1.) 4x1 MUX




```
entity mux4x1 is
  port (a: in std_logic_vector (3 down to 0);
        s: in std_logic_vector (1 down to 0);
        y: out std_logic);
end mux4x1;

architecture Behavioral of mux4x1 is
  begin
    process (s,a)
    begin
      case s is
        when "00" => y <= a(0);
        when "01" => y <= a(1);
        when "10" => y <= a(2);
        when "11" => y <= a(3);

        when others => y <= a(3);
      end case;
    end process;
  end behavioral;
```

2.) 8x1 MUX

entity a → 7 down to 0
s → 2 down to 0

process → for s
8-case statements "000" to "111"
outputs → a(0), a(1) . . . , a(7)

Teacher's Signature : _____

Date _____

Expt. No. _____ Page No. _____

with the use of if-else

```
process (s, a)
 begin
    if (s = "000") then y <= a(0);
 elsif (s = "001") then y <= a(1);
                .
                .
                .
  else  y <= a(7);
    end if;
end process;
```


Teacher's Signature : _____

SECTION A

Attempt all questions (objective/fill in the blank) **(10 x 1)**

Q1/co1 _____ declaration defines the bonding of one architecture body with many architecture bodies.

Q2/co1 A set of concurrent assignments represents _____.

a.) structure of entity **b.)** behavior of entity

c.) dataflow of entity **d.)** architecture of entity

Q3/co1 The structural style of modeling has two sections and those are

a.) declarative, statement **b.)** statistical, declarative

c.) behavior, dataflow **d.)** structure, dataflow

Q4/co1 _____ consists of a sensitivity list and it is invoked whenever there is an event on any signal in this list.

a.) Process **b.)** Signal **c.)** Variable **d.)** Clause

Q5/co1 Both sections of structural architecture body are separated by keyword_____.

a.) begin **b.)** process **c.)** variable **d.)** signal

Q6/co2 The correct way of constant declaration is

a.) constant rise_time : time:= 10ns; **b.)** constant rise_time ; time:= 10ns;

c.) constant rise_time : time:= 10ns: **d.)** constant rise_time : time:= 10ns:

Q7/co2 The values that are legal for a std_logic data object are

(i) 0 (ii) 1 (iii) H (iv) Z

a.) i and ii **b.)** ii and iii **c.)** i, ii and iv **d.)** All

Q8/co2 The two types of composite data types are

a.) scalar, vector **b.)** access, file

c.) array, record **d.)** scalar, constant

Q9/co2 _____ statement selects one of the branches for execution based on the value of the expression.

a.) if **b.)** wait **c.)** case **d.)** for

Q10/co2 Variable Assignment uses _____ operator and Signal assignment uses _____ operator.

a) <= and <= **b)** <= and := **c)** := and := **d)** := and <=

SECTION B **(Attempt any one part)**

Q1/co1(a) Write the syntax and explain with an example for following: **(2.5 x 2)**

(i) for – iteration scheme

(ii) if - statement

Q1/co1(b) Write a short note on Composite types used in VHDL. **(5)**

Q1/co1(c) What is a Package? How it is used in VHDL? Explain with syntaxes involved along with an example. **(5)**

OR

Q2/co1(a) Write the syntax and explain with an example for following: **(2.5 x 2)**

(i) while – iteration scheme

(ii) wait - statement

Q2/co1(b) Write a short note on sequential statements used within a process in VHDL. **(5)**

Q2/co1(c) List various scalar types used in VHDL. Define them with their range of values and also give 2 examples of each type literal. **(5)**

SECTION C **(Attempt any one part)**

Q3/co2(a) Write a VHDL code (Entity-declaration and behavioral modeling) for Full-Subtractor. **(7)**

Q3/co2(b) Write a gate-level VHDL code for 2x4 Decoder. **(8)**

OR

Q4/co2(a) Write a VHDL code (Entity-declaration and behavioral modeling) for 3-bit Binary parallel Adder. **(8)**

Q4/co2(b) Write a VHDL code for J-K flip flop. **(7)**

--- END ---