Unit-1

Camlin Page Date

* VLSI Design & Technology *

VLSI => Very Large scale Integrated circuits.

Depending upon no. of transistors implemented on single chip, digital IC's are classified as,

Technology	No of x'stors
(1) SSI → Small scale Integration	=> 1 to 10
eg. Basic gates IC's	
(2) MSI → Medium scale Inte^n	=> 10 to 500
eg. Multiplexers & flip flops, registers	
(3) LSI → Large scale Integration	=> 500 to 20,000
eg. PLD, PAL, PROM	
(4) VLSI → Very Large scale Inte^n	=> 20000 to 10,00,000
eg. FPGA, CPLD	
FPGA => Field programmable Gate Arrays	
CPLD => Complex programmable Logical Devices	
(5) ULSI → Ultra Large Scale Integration	=> >= 10,00,000
eg. large memories, large microprocessors.	

Unit- I

"VHDL Modeling"

=> VHDL stands for Very High Speed Integrated Circuits Hardware Description Language. (VHSIC - HDL)

=> HDL describes the hardware for the simulation, modeling, testing, design and documentation.

=> Portable Language :-
The source code in HDL for any design can be implemented on any device which is supported by synthesis tool.

=> Standard Language :-
We can ~~very~~ verify functionality of same VHDL code using different s/w tools such as xilinx, Actel, Altera etc.

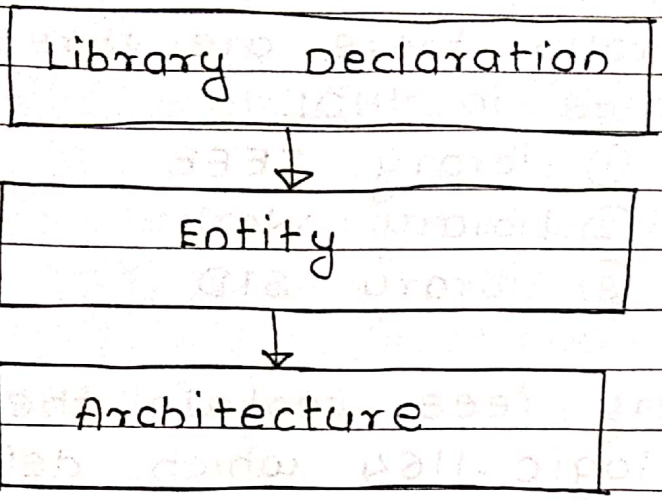
=> VHDL is easy to read, modify and structure.

=> Device and technology independent :-
With the same VHDL design, we can target many devices.

=> Disadvantages :- For analog elex. VHDL is not yet standardized, poor memory utilization, slower, repetitive logic etc.

* Structure of VHDL code :-

Structure of VHDL code consists of three sections,

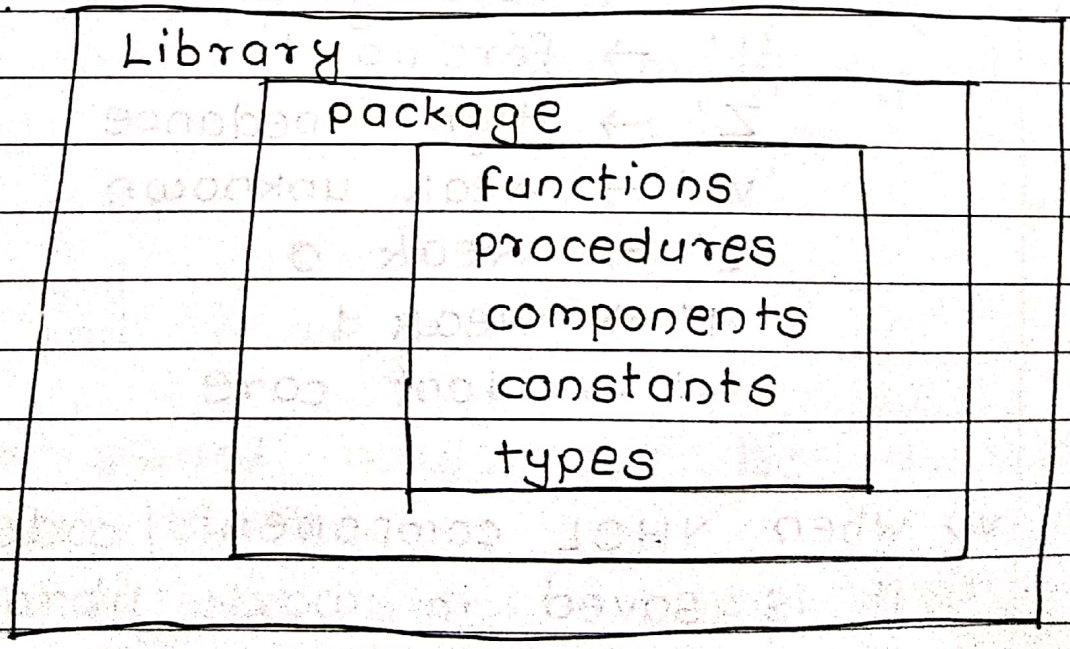


a) Library declaration :-

i) Library is set of functions which are used in program.

ii) Library code can be written in the form of functions, procedures and components and kept in the form of packages.

eg.



iii) Syntax

```
Library Library_name;
use
Library_name.package_name.part;
```

iv) Generally, there are three types of libraries in VHDL.

1. Library IEEE
2. Library Work
3. Library STD

v) Library ieee contain the package std_logic_1164 which defines nine logic values and associated overload functions.

eg.
```
library ieee;
use ieee.std_logic_1164.all;
```

Package std_logic_1164 consists of,

- 'U' → uninitialized
- 'X' → forced unknown
- '0' → forcing '0'
- '1' → forcing '1'
- 'Z' → High impedance
- 'W' → weak unknown
- 'L' → weak 0
- 'H' → weak 1
- '-' → dont care

vi) When VHDL component / code is compiled, it is saved in work library by default.

vii) Standard library has two packages, these are standard and textio.

viii) Package standard contains definations of predefined types and functions of language.
This packet contains following types- boolean, bit, character, integer, real, time, string, file_open_kind and file_open_status, natural, positive etc.

ix) Package textio contains declaration of types and subprograms that support formatted I/O operations on text files. It contains line, text, side, width etc.

b) Entity :-

→ Entity gives specification of input and output.

→ syntax :-

```
entity name is
port (
port_name : signal_mode signal_type;
port_name : signal_mode signal_type;
    :
    :

);
end name;
```

→ The signal mode can be in, out, inout, buffer.
In and out are unidirectional pins while inout is bidirectional.

Buffer → when output signal used internally in the design, buffer is used

→ The type of signal may be bit, std_logic, integer etc.

c) Architecture :-

Architecture is description of working of ckt design.

syntax

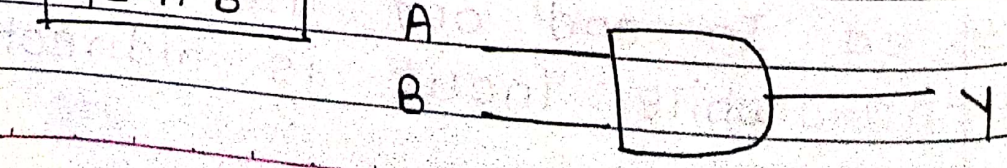
```
Architecture  arch^r_name  of entity_name is
        [signal & constant declaration]
                optinal
begin
        [ code ]
end   arch^r_name ;
```

eg. Write VHDL code for and gate.

=> AND Gate.

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Y = A·B



```
library { library ieee;
          use ieee.std_logic_1164.all;

Entity  { entity andgate is
          port (
                A,B : in std_logic;
                  Y : out std_logic );
          end andgate;

Arcbit  { architecture arch_and of andgate is
          begin
                Y <= A and B;
          end arch_and;
```

eg.2 Write VHDL code for Half adder.

⇒

a	b	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Kmap ⇒ for sum → S

b \ a	0	1
0	0	(1)
1	(1)	0

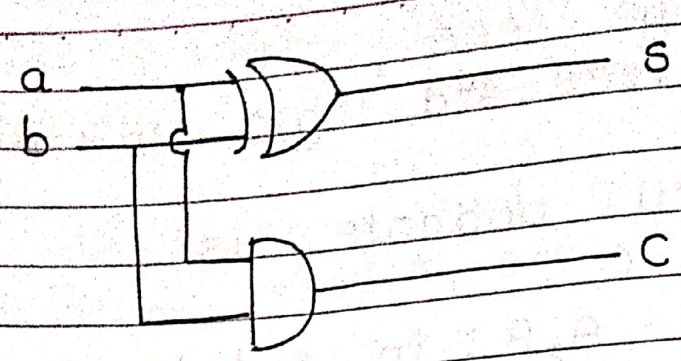
for carry → C

b \ a	0	1
0	0	0
1	0	(1)

$S = a\bar{b} + \bar{a}b$

$= a \oplus b$

$C = a \cdot b$




```
library ieee;
use ieee.std_logic_1164.all;

entity ha is
port(
    a,b: in std_logic;
    s,c: out std_logic);
end ha;

architecture hadd of ha is
begin
    s<= a xor b;
    c<= a and b;
end hadd;
```

* **Identifier :-**

- Identifier can contain alphabets, numbers and underscore '_'.
- It must be alphanumeric.

ie. abc1 → allowed
abc_123 → allowed
a12_3 → allowed
1a } Not allowed
1_abc }

Date / /

-> Identifier is user defined name for signal, constant, variable, entity, function architecture etc.

-> lower case and upper case letters are identical ie. VHDL is not case sensitive language.

-> No space are allowed

-> first character must be letter and last character should not be '_'. underscore.

- reserved words ie. keywords can not be used as ~~key~~ identifiers.

* Data objects :-

VHDL has three types of data objects

1. constant
2. variable
3. signal

1. constant :-

constant is used to assign the default values. constant value can not be changed once it is defined in design. constants are declared in archᵗ before begin.

Syntax :-

```
CONSTANT name : type := value ;
```

eg.

```
constant a : bit := '0';
constant bc : integer := 8;
constant delay : time := 10ns;
```

② Variable :-

→ A variable has single current value.
→ All value assignment to variable occurs immediatly.
→ The value of variable may change.
→ variables are used for local storage in process statement and subprogram.

syntax :-

```
variable name : type := intial-value ;
```

eg.

```
variable p : bit := '0' ;
variable A : integer range 0 to 100 ;
variable x : std-logic-vector (2 downto 0)
                := "000" ;
```

③ Signals :-

- signals are used to pass values in and out of the circuits.
- signals can be declared in entity, architecture declaration and package declaration.

→ Signals are communication media bet' entities.

- signal declared in entities can be shared in all entities and are called as global signals

syntax :

```
signal name : type := initial-value ;
```

eg. signal vcc : bit := '1';
signal gnd : std_logic := '0';
signal a : std_logic_vector (3 downto 0
:= "0000";

* Comparision bet^n variable & signal.

Variable	Signal
① value of variable updated immediatly, after execution of variable assignment statement.	② value of signal updated after an amount of time delay.
② It is declared under process.	② It is declared in architecture before begin statement.
③ Occupy less storage than signal	③ occupy more storage than variab
④ It has only two properties. i) type ii) value	④ It has three properties. i) type ii) value iii) time

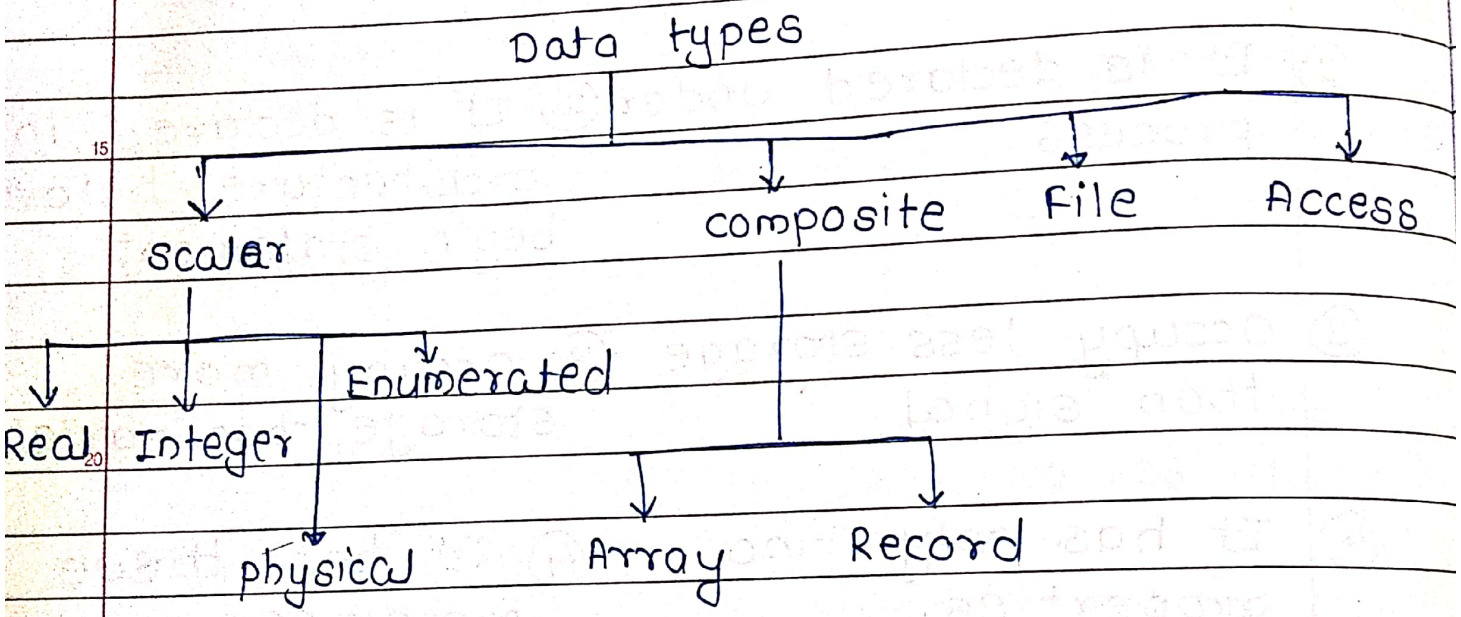
* Data Types :-

- Data types specify the characteristics of the object.
- Type declaration consists of name of the type and range of type.
- Type can be declared in entity, package, architecture, subprogram section.

syntax

type type_name is type_mark;

- VHDL consists of following datatypes.



i) * Scalar types :-

- scaler types object can hold only one value at a time.
- Scaler can have multiple ~~object~~ values, but an object defined to scalar value has only one value.
- Scalar has following types -

single bit assigning => '0', '1'
multiple bit -11- => "0010", "1011"

a) Integer types
b) Real types
c) Physical types
d) Enumerated types

- Integer types are same as math. integer having value bet^n -2^{31} to 2^{31}.
- Real types have no. ranging from -1.0×10^{38} to 1.0×10^{38}.
- Physical types are used to represent physical quantities such as distance, current, time etc.

eg.

```
type current range from 0 to 1000000000
units
    na;
    ua = 1000 na;
    ma = 1000 ua;
     a = 1000 ma;
end unit.
```

- Enumerated consists of identifiers and single character literals.
ie. bit ('0', '1');
Boolean (false, true)
Std_logic ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-')

- User defined enumerated type is,
type color is (red, green, yellow);

ii) Composite types :-

Composite types consist of collection of values. It has two types.

1. Array
2. Record.

* Array :-

- Array consists of multiple values of same type under single identifier.
- one or more dimensions.
- Values are referenced by indices.
- Indices type must be integer or enumerated.

syntax

```
TYPE array_name is array (range) of type;
```

eg.

```
type matrix is array (row, column) of boolean;
type byte is array (7 downto 0) of bits;
```

- Array subtype is subset of array.

syntax

```
subtype name is (array_name range );
```

eg.

```
type data is array (natural <>) of bit;
subtype low is (data range 0 to 7);
subtype high is (data range 8 to 15);
```

* Record :-

- Record contains element of different data types.
- Record in VHDL is similar to structure in C language.

Syntax

```
type <record_name> is record
    record defination;
    .
    .
    .
end record;
```

eg. student's Record arch^r.

```
type student is record
    Roll_no : integer range 1 to 100;
    name : string;
    marks : integer range 1 to 100;
    Passed : boolean;
end record;
signal s1, s2, s3, s5 : student;
```

- Elements of record can be accessed by using . (dot) operator.

eg.

```
s1.Roll_NO <= 10;
s1.name <= "kishor";
s1.marks <= 90;
s1.Passed <= TRUE;
```

* Access Type:-

- Access type is an address of specific object.
- Access type is similar pointer in C.
- Access type is used for dynamic nature object. eg. queue and stack.
- only variables can be declared as access type.
- Access type can be used in sequencial processing.
- For access type two predefined function are available.
    a) New
    b) deallocate

- New allocates of memory of size of object in byte.
- Deallocate takes access values and return the memory block to system.

syntax

type name is access ;

eg.

type tmp_name is access

* File Type :-

- File type allows declaration of file object
- file object is subtype of variable object type.
- A file object can be read from, written to and checked for the end of file only with special procedures & functions
- Every file ends with end_of_file mark.

eg.

read (file, data) => procedure allows read from file.

write(file, data) => procedure allows write to files.

endfile(file) => function return true value if file currently at end-of-file.

subtype :-

subtype is used to define subtype of type.

eg. type Number is range 00 to 99;
subtype mid_no is Number range 14 to 17;

* concurrent statements :-

VHDL consists of following interconnected concurrent statements.

i) concurrent signal assignment
ii) Block statement
iii) Generate statement
iv) process statement
v) component Instantiation statement.

(i) concurrent signal Assignment :-

- simple concurrent signal assignment is given as,

syntax

target <= expression ;

ie. value of expression is assigned to target.

eg.

z <= a and b;

ie. logical anding of a and b assigned to signal z.

- signal assignment statements are sensitive to expression that are to the right of <= symbol.

- conditional concurrent signal assignment is given as,

syntax :-

target <= booleanexpression when condition else

expression 2 ;

- while executing when statement each condition is tested in the order in which it is written.

- The value of expression whose condition is true will be assigned to the target.

- If none of the condition are true, value of expression with last else will be assigned to target.

eg.

```
Z <= A when a = "00"
     else
     B when a = "01"
     else
     C when a = "11"
     else
     D ;
```

- selected concurrent signal assignment can be done as,

syntax

```
with expression_choice select
target <= expression1 when choice1,
          expression2 when choice2,
          expression3 when choice3,
          .
          .
          .
          expressionN when others;
```

eg. (4:1 mux)
with control select

```
Y <= I0 when "00",
     I1 when "01",
     I2 when "10",
     I3 when "11",
     '0' when others;
```

- other clause is optional and it must be last choice.

* Block statement :-

- Using block statement we can group the model area logically.
- signal, types, constants etc. are declared in block and these are local to block & can not be used outside the block.

syntax

```
label : block
    [block declaration items]      <- optional
begin
        concurrent statements;
        .
        .
        .
end block [label];
```

→ lable is name for block
→ The block can be nested.
→ Guarded block contains a guard expression which can enable and disable

driver inside the block.

- The guard expression is boolean expression.

→ when guard expression is true, driver contained in block are enabled, and when false, the drivers are disabled.

eg.

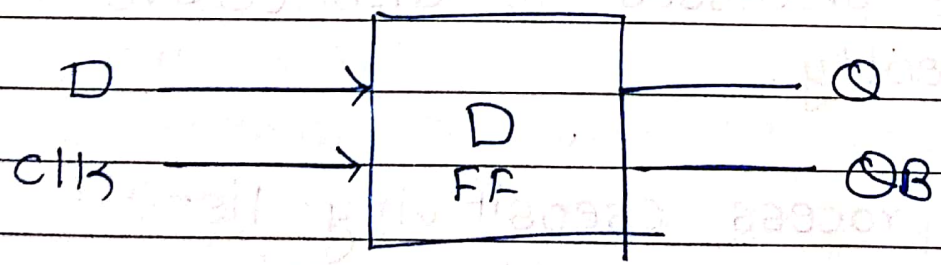
```
ABC : block (clk = '1').
begin
    z <= guarded not x;
end block ABC;
```

- Here guarded expression is clk = '1'.

syntax

```
Label : block (guard expression)
begin
    concurrent statements;
    .
    .
    .
end block Label;
```

* Write VHDL code code for D. flip flop



when clk = 1, Q = D

QB = $\bar{D}$

```
entity DFF is
port ( D, clk : in bit;
       Q, QB : out bit );
end DFF;

architecture arch-D of DFF is
begin
    B : block ( clk = '1' )
    begin
        Q <= guarded D;
        QB <= guarded not(D);
    end block B;
end arch-D;
```

Process statement :-

- Process statement contains only sequential statements.

- All statements in process are executed sequentially ie. order of statement is important.
- All the processes in architecture execute concurrently.

Syntax

```
process (sensitivity list)
        Declaration part
begin
        sequential statements
        :
end process;
```

=> Sensitivity List :-

- sensitivity list is list of signals to which process is sensitive.
- sensitivity list contains the signals which are cause of process statement execution.

If value of any sensitivity list signal changes, the statements inside the process statements are executed.

- sensitivity list should not contain wait statement.

→ Process should have either sensitivity list, or wait statement.

=> Declaration part :-

- It consists of, declaration of type, variables, constants, subprogram etc.
- Process never stops, it repeats forever, unless suspended.
- To suspend the process either wait statement or sensitivity list is used

=> wait statement :-

- wait statement is only used in process statement to suspend the process.
- There are three types of wait statements.

(1) wait on sensitivity_list
(2) wait until boolean_expression
(3) wait for time_expression

We can also combine all three types in one statement.

ie.

wait on sensitivity_list until boolean_expression for time expression ;

eg. a) process with wait

```
process
    begin
        if a > c then
            y <= '1';
        else
            y <= '0';
        end if;
        wait on a, c;
    end process;
```

b) process with sensitivity list.

```
process (a,b)
    begin
    if a>b then
        y<='1';
    else
        y<='0';
    end if;
end process;
```

- In above example if event occurs on a,b then execute the statements (ie. when event occurs on a,b, process resumes execution).

- If wait statement is last statement in process, the process resumes execution from first statement.

- examples of wait statements

① wait until clk = '1' => wait on boolean expression

② wait on x, y, z => wait on sensitivity list.

③ wait for 10ns => wait on time expression

④ wait on a for 100ns => wait on sens. list & time

⑤ wait until a > 10 for 50 ns => wait on boolean & time expression.

⑥ wait on a until b > 50 => wait on boolean expression & sens. list.

* Generate statement :-

- Generate statement is used to select concurrent statement conditionally.
- It is used to create multiple copies or replication of process, component or blocks.
- Generate statement has two types.
  - ① For .... generate
  - ② if .... generate

① For .... generate :-

- It generates multiple copies of block, processes, or components.

syntax

```
label : for identifier in range generate
            concurrent statements;
            :
            .
        end generate label;
```

- range must be computable integer.
- for_generate creates new local integer variable with the name identifier.

- First value of range is assigned to identifier. and each concurrent statement is executed once. Identifier is assigned next value in range of each concurrent statement is executed once more.
- This process is repented up to last value in range.
- After 'end generate' statement the loop identifier is deleted.

(2) If ... Generate :-

- 'If... generates creates zero or one copy conditionally.

Syntax :-

```
label: if expression generate
        concurrent statements;
        .
        .
        .
    end generate label;
```

- If the expression is true, then concurrent statements are executed once, otherwise no execution of concurrent statements.

eg.

```
label : if a = '0' generate
        val <= val + 1;
    end generate label;
```

* Component Instantiation statement :-

- component is predesigned, preanalyzed, precompiled entity-architecture pair.
- Component is subsystem.
- component can be defined in packages, entity, architecture or block statements
- components are declared before begin statement.
- components must be declared before instantiated.

syntax :- component declaration :-

```
component component_name
    port (
            port list;
          )
end component;
```

eg.

```
component HAdder
port (
        a, b : in std_logic;
        s, c : out std_logic );
end
```

- component name should be same as entity name.
- component instantiation statement is used to build a net list in VHDL by referencing previously defined hardware components in current design.

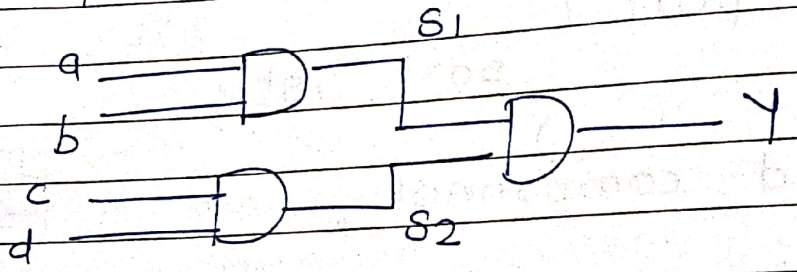
- In component instantiation previously designed, compiled subsystems are called in main program.

Syntax

Instance_label : component_name
port map (expression, ...);

- port map connects each port of component to signal valued expressions in current entity.

** ex. design VHDL code for fig. using components.



a) Design of and gate :-

```
entity and1 is
port
    (x, y : in bit;
     z : out bit);
end and1;

architecture arc_and of and1 is
begin
    z <= x and y;
end arc_and;
```

b) Design of main ckt

```
entity main is
port (
      a, b, c, d : in bit;
            Y : out bit);
end main;

architecture arc_ckt of main is
   component and1
   port (x, y : in bit;
         z : out bit);
   end component;

   signal s1, s2 : bit;
begin

   G1 : and1 port map (a, b, s1);
   G2 : and1 port map (c, d, s2);
   G3 : and1 port map (s1, s2, Y);
end arc_ckt;
```

— x —

** port mapping can be also done in following way,

```
G1 : and1 port map (x => a, y => b, z => s1);
G2 : and1 port map (x => c, y => d, z => s2);
G3 : and1 port map (x => s1, y => s2, z => Y);
```

* Styles of Modeling :-

The ways of description of functionality of design are called as styles of modeling.

It has four types.

i) Data flow modeling
ii) Behavior Modeling
iii) Structural Modelling
iv) Mixed type modeling.

i) Data flow modeling :-

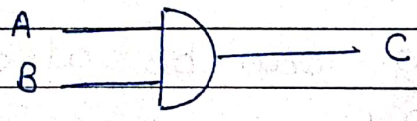
- This model needs boolean expressions for design.
- flow of data is expressed thro' concurrent signal assignment statements.

eg.

```
target <= expression_boolean;
```

- Each statement executed when input signal changes its value.

eg. Design of AND gate.




```
entity andgate is
port ( A,B : in std_logic;
       C : out std_logic);
end andgate;

architecture df_and of andgate is
begin
    C<= A and B;
end df_and;
```

ii) Behavioral Modeling :-

- Behavioural modeling uses sequential statements.
  eg. if else, if, when, case etc.
- It is not gate level implementation of design like dataflow modeling.
- This modeling needs truth table as design specification.
- It is also known as high level description language.

eg.

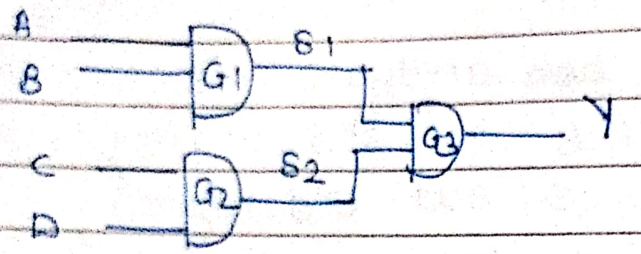
Design of AND gate using behavioral modeling.

```
entity ~~beh~~-andgate is
port (A,B : in std_logic;
        C : out std_logic);
end andgate;

architecture beh_and of andgate is
begin
    process (A,B)
    begin
        if (A='1' and B='1') then
            C<='1';
        else
            C<='0';
        end if;
    end process;
end beh_and;
```

iii) Structural Modeling :-

- In this modeling, precompiled, predefined, preexecuted VHDL code are used as components for design.
- This method is hierarchical.
- The components can be taken from libraries or current directory.
- This design needs logical diagram for design.
- Each component can be individually simulated.

eg. Write structural code for ckt.



We have already simulated and gate, now we can take it as component.

```
entity ckt is
port (A,B,C,D : in std_logic;
            Y : out std_logic);
end ckt;

architecture str_ckt of ckt is
component andgate
port (A,B : in std_logic;
         C: out std_logic);
end component;
signal S1, S2 : std_logic;
```

```
begin
G1 : andgate port map (A,B,S1);
G2 : andgate port map (C,D,S2);
G3 : andgate port map (S1,S2,Y);
end str_ckt;
```

* Sequential statements :-

- Sequencial statements are executed in the order in which they are written.
- Sequential statements can appear only in process or subprogram.

(1) IF statement

syntax 1	ie. if condition is true all statements will be executed.

if condition then
statements ;
end if;	

syntax 2 – if — else

if condition then
statement1 ;
else
statement 2 ;
end if ;
	ie. if condition is true, statement 1 executed else statement 2 will be executed.

& IF we have multiple conditions, then we can use nested if - else.

syntax - 3 Nested if else.

```
if condition1 then
    statement1;
elseif condition 2 then
    statement 2;
elseif condition 3 then
    statement 3;
    :
    :
else
    statement;
end if;
```

eg.

```
process (A, B, C, X)
begin
    if (X = "000") then
        Z <= A;
    elseif (X = "001") then
        Z <= B;
    else
        Z <= C;
    end if;
end process;
```

(2) Case statement :-

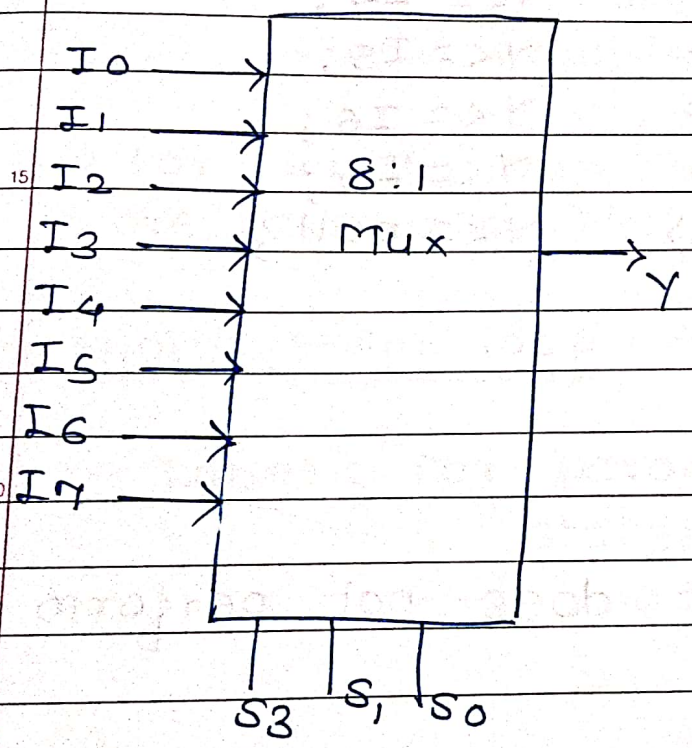
Case statement produces parallel logic whereas if produces priority based logic. All conditions of any expression are covered using when statement.

Date / /

syntax

```
case expression is
    when choice1 => statement1 ;
    when choice2 => statement2 ;
    when choice 3 => statement3 ;
    when others => statements ;
end case ;
```

eg. Design 8*1 mux using case statement



S

S_2	S_1	S_0	Y
0	0	0	I0
0	0	1	I1
0	1	0	I2
0	1	1	I3
1	0	0	I4
1	0	1	I5
1	1	0	I6
1	1	1	I7

```
entity mux_81 is
port (
    I0, I1, I2, I3, I4, I5, I6, I7 : in std_lo
    Y : out std_logic ;
    S : in std_logic_vector (2 downto
end mux_81 ;
```

```
architecture df_mux of mux_81 is
begin
process (S, I0, I1, I2, I3, I4, I5, I6, I7)
begin
        case S is

when "000" =>   Y <= I0;
when "001" =>   Y <= I1;
when "010" =>   Y <= I2;
when "011" =>   Y <= I3;
when "100" =>   Y <= I4;
when "101" =>   Y <= I5;
when "110" =>   Y <= I6;
when "111" =>   Y <= I7;
when others =>  Y <= null;
        end case;
    ~~NUH stal~~
    end process;
end df_mux;
```

* => Null statement does not perform any action.

3) Loop statement :-

Loop statements are used for repeatative execution of sequential statements.
It has two types.
1) while loop statement
2) for loop statement.

Camlin Page
Date / /

Syntax :- while loop statement

```
Label : while condition loop
        statements;
        end loop Label;
```

eg.
```
process (x)
begin
    L1 : while y <= 5 loop
        z(i) <= x(i+4);
        i <= i+1;
    end loop L1;
end process;
```

⇒ For while loop sequence of statements are executed till condition is true.

Syntax :- For loop statement

```
Label : for parameter in range loop
        statements;
        end loop Label;
```

eg.
```
Fact := 1;

L1 : for number in 2 to 10 loop
        fact := fact * number;
    end loop L1;
```

⇒ The loop will execute for each value in range.

④ Next statements :-

Syntax

i) Next

ii) next loop-label when condition

Next statement is used inside the loop and it is used to skip the execution of remaining statements in current iteration of loop and execution starts from next iteration of loop.

eg.

```
a1: for x in 1 to 10 loop
        sum := sum + 10;
        if (sum = 100) then
            next;
        else
            null;
        end if;
    end loop a1;
```

⑤ Exit statement :-

Exit statement totally terminates execution of loop.

Syntax

i) exit

ii) exit loop-label when condition

eg.

```
L1: for x in 1 to 5 loop
        sum := sum + 10;
        if (sum > 30) then
```

```
exit L1;
end loop L1;
```

(5) Report statement :-

- Report statement is used to print or display the string or message and severity level to be reported to simulated for proper action.
- Severity level has following standard values.
  - i) note
  - ii) warning
  - iii) error
  - iv) failure

Assert statement

- Normally, report statement is used with assert statement.
- If the condition in assert is false during simulation, a message of certain severity is sent to user.

eg.

syntax

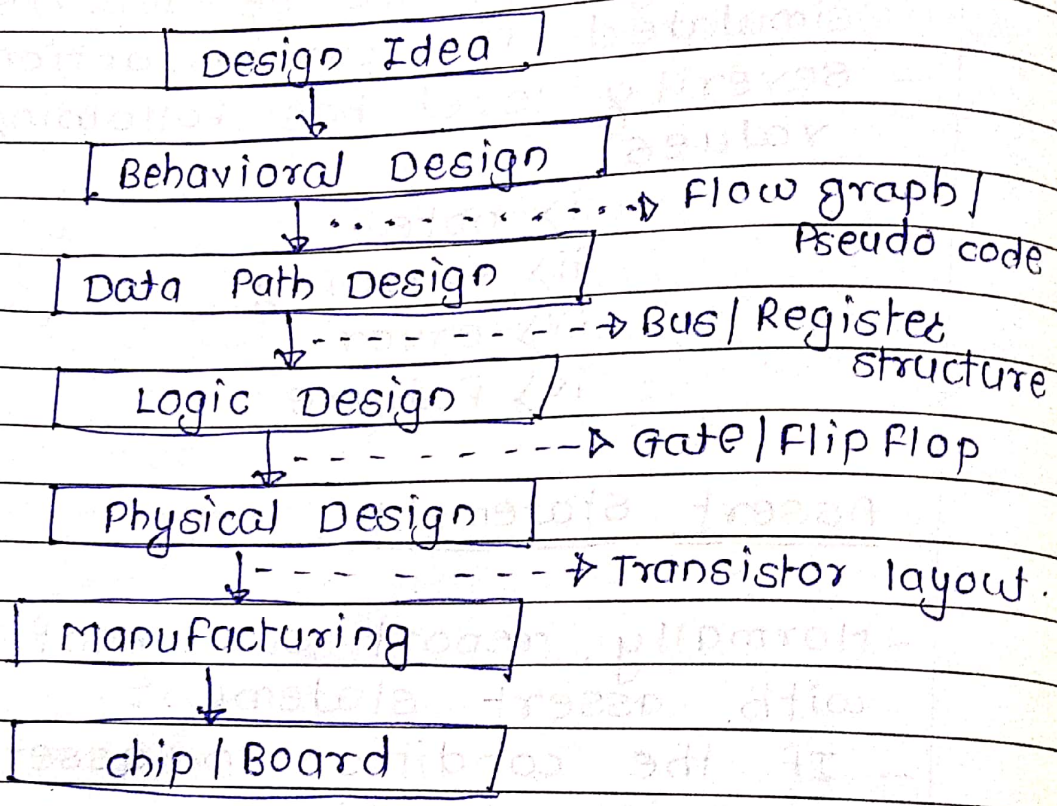
```
Assert (condition)
Report (message)
Severity (level)
```

eg.

```
process
begin
```

```
assert not (S='0' and R='0')
report (" S and R are both low, Not
                valid inputs " ;
severity error;
end process;
```

* VLSI Design Flow :-



a) Behavioural Design :-

- Specify the functionality of design in term of its behaviour.
- We can specify it using
  - a) boolean expression
  - b) finite state machines
  - c) high level algorithms
- Need to synthesized into more detailed specification for hardware realization.

b) Data Path Design :-

- Generate a netlist of register transfer level components like registers, adders, multipliers, multiplexers, decoder etc.
- A netlist is directed graph, where vertices indicates components and edges indicates interconnections.

c) Logic Design :-

- Generates netlist of gates, or flip flops or standard cells.
- A standard cell is predesigned circuit module like gates, flip flops, multiplexers etc at the layout level.
- Various logic optimization techniques are used to obtain cost effective design.
- The requirement for optimization is,
  a) minimum no of gates
  b) minimum gate delay ie. delay
  c) minimum signal transition
     ie. dynamic power

d) Physical design & manufacturing :-

- Generate final layout that can be sent for fabrication.
- The layout contain large no of regular geomentric shapes corresponding to diff fabrication layers.
- Finally mapping of gatelevel netlist on field programmable gate Array devices is done (FPGA).

b) Data Path Design :-

- Generate a netlist of register transfer level components, like registers, adders, multipliers, multiplexers, decoder etc.
- A netlist is directed graph, where vertices indicates components and edges indicates interconnections.

c) Logic Design :-

- Generates netlist of gates, or flip flops or standard cells.
- A standard cell is predesigned circuit module like gates, flip flops, multiplexers etc at the layout level.
- Various logic optimization techniques are used to obtain cost effective design.
- The requirement for optimization is,
  a) minimum no of gates
  b) minimum gate delay ie. delay
  c) minimum signal transition
     ie. dynamic power

d) Physical design & manufacturing :-

- Generate final layout that can be sent for fabrication.
- The layout contain large no of regular geomentric shapes corresponding to diff fabrication layers.
- Finally mapping of gatelevel netlist or field programmable gate Array devices is done (FPGA).

* Operators in VHDL :-

VHDL has following types of operators.

1. Assignment operator
2. Shift operator
3. Logical operator
4. Relational operator
5. Arithmetic operators
6. concatenation operator

① Assignment operators :-

a) signal/variable assignment :- (<=)

syntax

target <= value/expression;

eg.

b <= a and c;

b) signal initialization :- (:=)

variable a: integer := 10;

signal b: std_logic := '0';

② Shift Operator

SLL → Shift left, right most bits replaced with zeros

SRL → Shift Right, left most bits replaced with zeros.

SLA → Shift left arithimatic

SRA → Shift right arithimetic

ROL → Rotate left

ROR → Rotate Right

Examples:

"1001" sll 2 => 0100, "1001" srl 2 => 0010
"0101" sla 2 => 0111, "1010" sra 2 => 1110
"0111" rol 2 => 1101, "1011" ror 2 => 1110

(3) Logical operators

① AND → Logical operators works on
② OR → predefined types, either Bit or
③ NAND → Boolean. The resul has the same
④ NOR → type as the type of operand (s)
⑤ XOR → Ex: z := x AND y;
⑥ XNOR → variables
⑦ NOT → c <= a AND b; -- signals

(4) Relational operators

= eq
/= not eq
< lt
<= lt or eq
> gt
>= gr or eq

Relational operators compare two operands of the same type and produce a Boolean type.
ex: z <= x <= y;
--- If x less than or equal to y, the Boolean result will be assigned to the signal z.

(5) Arithmetic operators

** Exponentiation
rem Remainder
mod Modulus
/ Division
* multiplication
- Subtraction
+ Addition
abs Absolute Value
- Unary Minus
+ Unary plus

(6) Miscellaneous Operators

& Concatenation Operator

Example:
"ABC" & "DEF" => ABCDEF
"1010" & "0101" => 10100101

Prof. K. B. Bhangale.

Subprogram :-

A subprogram defines a sequencial algorithm that performs a certain operation There are two types of subprogram.

i) Function
ii) Procedures

Functions are used to compute a single value. A function executes in zero simulation time.

Procedure can return zero or more values. A procedure may or may not execute in zero simulation time.

Function :-

- Functions are used to describe the frequently used sequencial algorithm that returns a single value.
- This value is return to calling program using return statement.
- Functions are used as resolution functions and datatype conversion functions.
- A pure function is function which return the same value each time the function is called with same set of input.

- An impure function is function whi return different value each time when it is called with same value of actual inputs.

- The parameter list describes the list of formal parameter for the function

- The parameter mode is always in. only constant and signal object can be passed in as parameter. Default object class is constant.

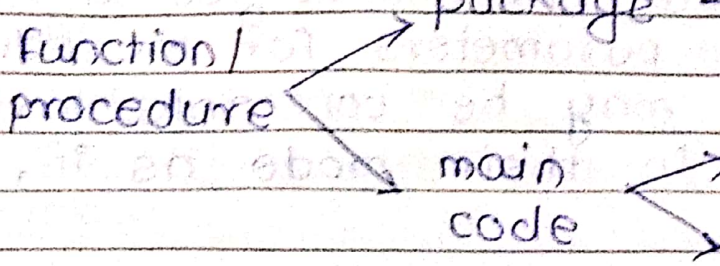
- An actual may be associated with position (first actual corresponds to first position, second actual corresponds to second position and so on).. or using naming associatio

Syntax

```
FUNCTION Function_name
[parameter list] RETURN data_type IS
    [declaration]
begin
    ( sequencial statements)
END function_name;
```

⇒ function is called with input valu

- Function can be located in package (for code reuse, code partitioning and code sharing purpose or it can be located in main code either in entity or architecture.



eg.

```
-- Function Body

FUNCTION clock_1 (signal s: std_logic)
  return Boolean is
  begin

    return (s'event and s='1')
  end clock_1;

-- Function call

IF  clock_1 (clk) then
      :
```

Procedures:-

- Procedure can return zero or more values using parameter of mode out and inout.

- The parameter list specifies a list of formal parameters for procedure. Parameters may be constants, variables or signal in their mode as in, out or inout.

- Syntax (procedure body)

```
procedure procedure_name (parameter list)
```

- procedures are envoked by procedure call. A procedure call can be either a sequencial statement or concurrent statement.

- If the procedure call is inside the process statement or another subprogram it is a sequencial procedure call statement; otherwise it is concurrent procedure call statement.

- Syntax (procedure call)

```
[label:] procedure_name (list of actuals
```

- A sequencial procedure call statement is executed sequencially with respect to the sequencial statements.

- A concurrent procedure call statement is executed whenever an event occurs on one of the parameter that is a signal or mode in or inout.

- Concurrent procedure call is equivalent to a process with sequencial procedure and wait that wait for an event on signal parameter of mode in or inout.

- A procedure body can have wait statement, while the function cannot.

eg.

```
type opcode is (Add, sub, Mul, Div);
procedure arith_op (A,B: in integer;
                    op : in opcode;
                    out1 : out integer) is
begin
   case op is

      when Add =>
                    out1 := A+B;
      when sub =>
                    out1 := A-B;
      when Mul =>
                    out1 := A*B;
      when Div =>
                    out1 := A/B;
   end case;
end arith_op;
```

Function	Procedure
Function can return only single value.	procedure function can return zero or more value
Function returns its own value.	procedure does not have return value.
Function does not contain wait statement.	procedure contains wait statement.
Function executes in zero simulation time.	procedure may or may not execute in zero simulation time
Function call is an expression and can be used in large expressions.	procedure call is sequencial or concurrent statement depending upon when it is used.

* Libraries :-

- Library contains set of utility packages

- There are 3 types of libraries.
  - Library ieee
  - Library work
  - Library std

Library ieee

- library ieee contains the package std_logic_1164.all which contain nine value logic types and its associated overloaded functions and utilities.

- package std_logic_1164 contains following types

```
std_logic is
  (
    'U' - uninitialize
    'X' - forcing unknown
    '0' - forcing zero
    'I' - forcing one
    'Z' - high impedance
    'W' - weak unknown
    'L' - weak 0
    'H' - weak 1
    '-' - don't care
  );
```

- It also contain fun° like and, or, nor, xor, xnor, not etc.

Library std

Library std package

standard package
↓
contain predefined types, functions of language, subtype

textio package
↓
contains subprogram which support formatted input output operation on textual file

Library work:-

present working library can be identified by keyword work in VHDL

- In VHDL standard work and std libraries are always visible . :. these two libraries do not have to be specified in VHDL code.

- to library and use clause are used to include library packages in VHDL code.

Packages :-

→ Package provides a convenient mechanism to store and share the declarations that are common across many design units.

→ A package is represented by,
a) Package declaration
b) package body.

a) Package declarations :-

i) A package declaration contains set of declarations that may be shared by many design units.

ii) It defines the items that can be made visible to other design units.
eg. function declaration.

iii) Syntax of package declaration is,

```
Package package_name is
-- type declaration
-- subtype declaration
-- subprogram declaration
-- signal declaration
-- constant declaration
-- file declaration
-- variable declaration
-- alias declaration
-- attribute declaration
```

-- component declarations
-- use clauses
end package_name;

iv) Items declared in packages can be access by other design units using library and use clause.

v) The set of common declarations may also include function and procedure declarations.

b) Package Body :-

i) A package body primarily contains the behaviour of subprograms and the values of deferred constants declared in a package declaration.

ii) Syntax :-

package body package_name is
-- subprogram bodies
-- complete constant declaration
-- subprogram declarations
-- type and subtype declaration
-- file and alias declarations
-- use clauses
end package_name;

iii) package name must be same as the name of its corresponding package declarations.

iv) A package body is not necessary if its associated package declaration does not have any subprogram or deferred constant declarations.

v) An item declared inside the package body has its scope restricted to be within the package body and its its item can not be made visible in other design.

vi) ∴ package body is used to store private declarations that should not be visible, while a package declaration is used to store public declaration that should be visible to other units.

* Test Bench :-

i) Test bench is the model which is used to verify functionality of hardware design.

ii) We can verify the functionality of design at each step using HDL synthesis based methodology.

iii) The test bench has three main purpose.
   a) to generate stimulus for simulation (waveforms).
   b) to apply this stimulus to the entity under test and collect the output response.
   c) to compare output response with expected values.

iv) A testbench is at highest level in the hierarchy of design. The testben instanticates the design under test.

v) There are five types of testbenches
   a) Stimulus only => contain only stimulus driver and design unde test, does not contain any resu verification.

   b) Full testbench => contain stimul driver, known good results an result comparision.

c) Simulator specific testbench :-
Testbench is written in simulator specific format.

d) Hybrid testbench :-
combines techniques from more than one testbench style.

e) Fast testbench :-
Testbench written to get ultimate speed from simulation.

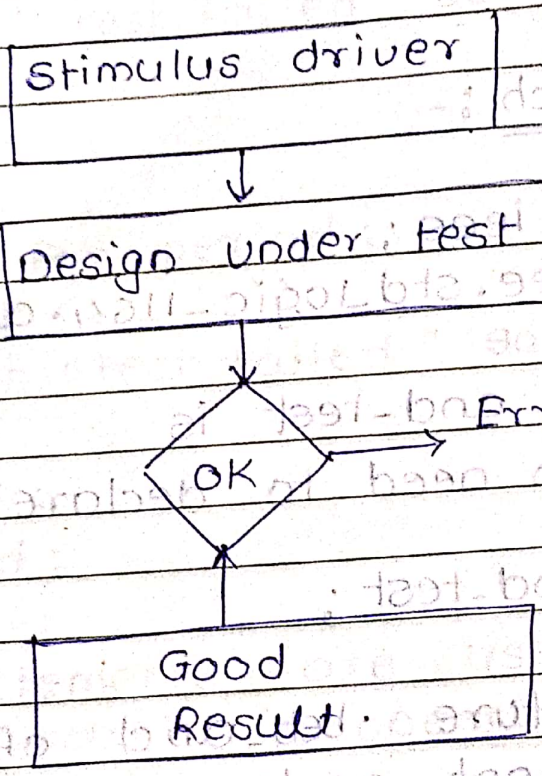


fig. test bench flow diagram

eg. Design the AND gate and verify the design of AND gate.

→

```
library ieee;
use ieee.std_logic_1164.all;
```

```
entity andgate is
    port (
            A,B : in std_logic;
            C : out std_logic);
end andgate;

architecture beh_and of andgate is
begin
    C <= A and B;
end beh_and;
```

Test Bench :-

```
library ieee;
use ieee.std_logic_1164.all;

entity and_test is
    -- no need to declare entity
end and_test;

architecture beh_and of and_test is
component and2
port ( A,B : in std_logic;
            C : out std_logic );
end component.
signal A,B,Y : std_logic;
begin

I1 : and2 portmap (A => A, B => B, C => y)
process
constant period : time := 40 ns;
```

```
begin
    A <= '1';
    B <= '1';
    wait for period;
    assert (y = '1')
    report "Test Failed" severity error;

    A <= '1';
    B <= '0';
    wait for period;
    assert (y = '0')
    report ("test failed" severity error;

    A <= '0';
    B <= '0';
    wait for period;
    assert (y = '0')
    report "test failed" saverity error;

end process;
end beh_and;
```

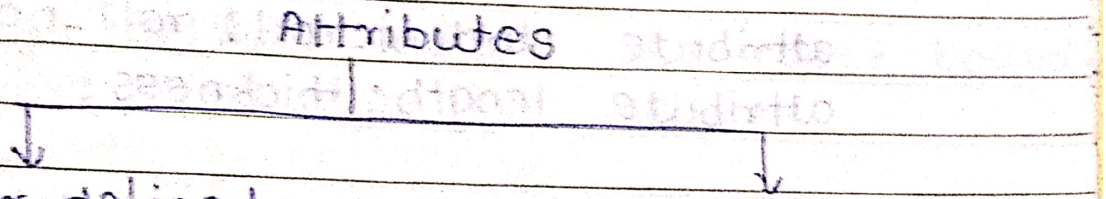
=> assert statements are used to verify that the circuit is operating correctly for each combination of input:

Attribute :-

An attribute is a value, function, type, range, signal or constant that can be associated with certain names within VHDL description.

→ These names could be entity name, an architecture name, label or signal.

→ Attributes have two types.



(A) User defined attribute :-

User defined attributes are constants of any type (except access or file type). They are declared using attribute declaration.

An attribute declaration declares name of attribute and its type.

An attribute can be declared as follow,

```
attribute attribute_name : value_type ;
```

eg.

```
type roll-no is
record
        x, y : integer;
end record;
type thickness is range 0 to 500
units
        micron;
end units;

attribute student_roll : roll_no;
attribute length : thickness;
```

→ User defined attributes are useful for annotating VHDL model with tool specific information.

(B) Predefined Attributes :-

There are five classes of predefined attributes.

i) value attributes :- these return const value
ii) function attribute :- these calls fun° and return a value
iii) signal attribute :- these creates new signal
iv) type attributes :- these return type name
v) Range attribute :- these return a range.

① Value attribute :-

Value type attributes are used to return the bound of a type.
If T is any scaler or subtype then

T'left => which returns the left bound of type or subtype.

T'Right => which returns the right bound of type or subtype

T'High => which return the upper bound of type or subtype

T'low => which return the lower bound of type or subtype

T'Ascending => which return the true value if type has ascending order otherwise returns false.

T'length => Return the length of array.

eg. type rollno is (0 to 100);

for above type,

rollno'left = 0;
rollno'right = 100;
rollno'high = 100;
rollno'low = 0;
rollno'ascending is true;

(11) Function attribute :-

These attributes represents functions that called to obtain a value. Normally function attributes are used to convert values from an enumeration or physical type to an integer type.

If T is a discrete type, physical type or subtype then;

T'pos(V) => return the position number of the value V in ordered list of value of T.

T'val(P) => return the value of type that corresponds to position P

T'succ(V) => return the value of parameter whose position is one larger than position of value V in T.

T'pred(V) => return the value of parameter whose position is one less than position of value V in T.

T'leftof(V) => return the value of the parameter that is to the left of value V in T.

T'Rightof(V) => return the value of the parameter that

right of value Y in T.

If S is signal object then

S'event => return true value if an event occured on signal S in current data.

S'active => return true if signal S is active in current data.

S'last_event => return time elapsed since the last event on signal S.

S'last_active => return the time elapsed since last time signal was active.

S'last_value => return the value of S, before the last event.

Signal Attributes :-

These attribute create new signals from the signals with which they are associated.

These attribute creates implicit signals, as compared to explicit signals that are created using signal declaratio

If S is signal object then,

S'delayed (T) => new signal that is the same type as signal s but delayed from s by time T.

S'stable (T) => return boolean signal that is true when signal s has not had any event for time T.

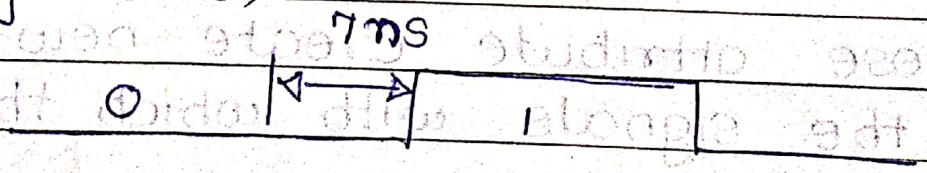
S'quiet (T) => create the boolean signal that is true when s has not been active for time T.

S'transaction (T) => creates a signal of type bit that toggles its value every time signal s becomes active.

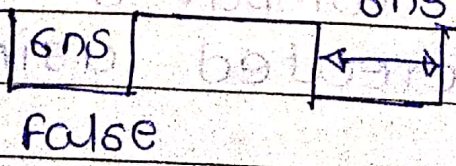
eg.

signal clock : bit;

clock'delayed (7ns)



clock'stable (6ns)



Type Attribute :-

If T is any type or subtype, then

T'base => returns base type of T.

This attribute can not be used in expressions ~~as such as~~ because it returns a base type, but it can be used as conjuction with other type.

eg.

```
type operation is (add, sub, mul, AND, OR);
subtype operation is arith_op range
                        add to mul;
subtype operation is logic_op range
                        AND to OR;

type operation is (add, sub, mul, or, and, not);

subtype arith_op is operation range add to
                        mul;
subtype logic_op is operation range
                        or to not;
```

Range Attribute :-

If A is constrained array object then,

A'range(N) => return the Nth index range of A

A'reverse_range(N) => return the Nth index range reversed.

eg.

variable wbus : std_logic_vector (7 down

wbus'range => returns range "7 down to

wbus'reverse_range => returns range "0

- Range attribute can be used in index constraint in for loop statem

eg.

```
for index in wbus'reverse_range loop
        :
end loop;
```

* Sequencial Circuits :-

- A digital ckt is called as sequencial ckt if value of output at any time depends upon value of input at that time and previous value of input / output.

- Atleast one feedback must be present between input and output.

- memory element is needed to store previous value of input or output.

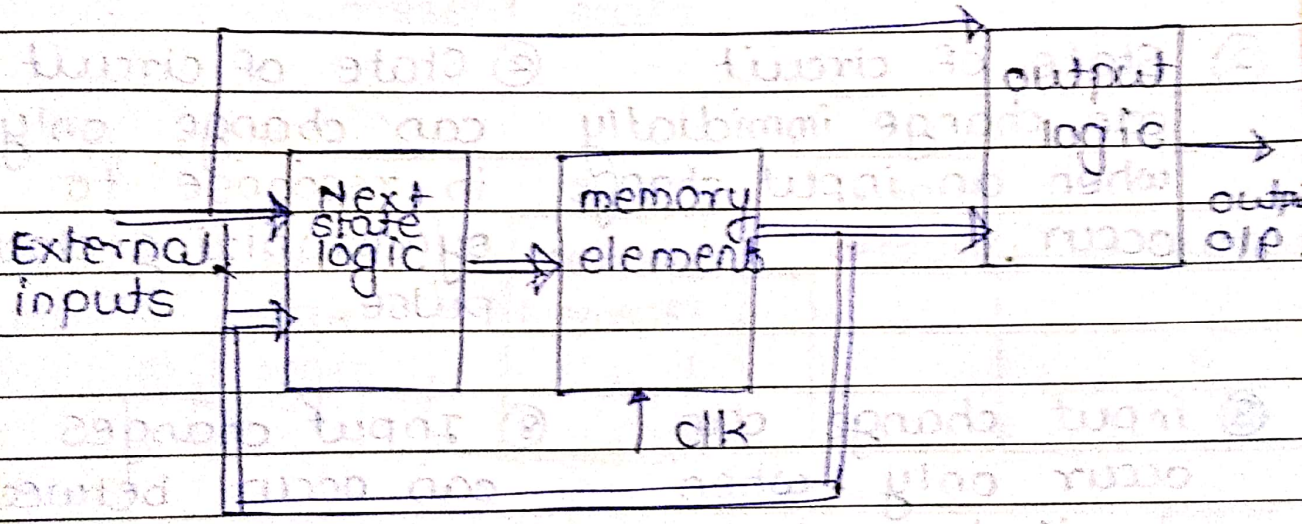


Fig. Sequencial ckt.

- Next state logic may be combination ckt which accept the input from external signals or output & generates signal for memory or output.

→ Memory element may be clocked or unclocked flip-flop.

- output logic may be combinational ckt which accept input from memory

and external input and generate external signal.

- Sequencial machines or ckts are classified in two types.
  1. Asynchronous circuits
  2. Synchronous circuits

Asynchronous circuit	Synchronous circuits
(1) Asynchronous ckts do not need clock.	(1) It needs clock.
(2) State of circuit can change immidiatly when an input change occur.	(2) State of circuit can change only in response to synchronizing clock pulse.
(3) input change can occur only when circuit is in stable state.	(3) Input changes can occur between clock pulses.
(4) Memory element is latch (unclocked flip flop) or gate ckts.	(4) Memory element is clocked flip-flop
(5) only one input can change at a time.	(5) No. of input can change at a time
(6) Design is difficult	(6) Design is simple

* Finite State Machines :-

Finite state machines are used for specific patterned sequence.
FSM can be classified in three classes.

1) class A
2) class B
3) class C

class A (Mealy FSM Machine)

In mealy machine output is function of
i) present input and
ii) present state.

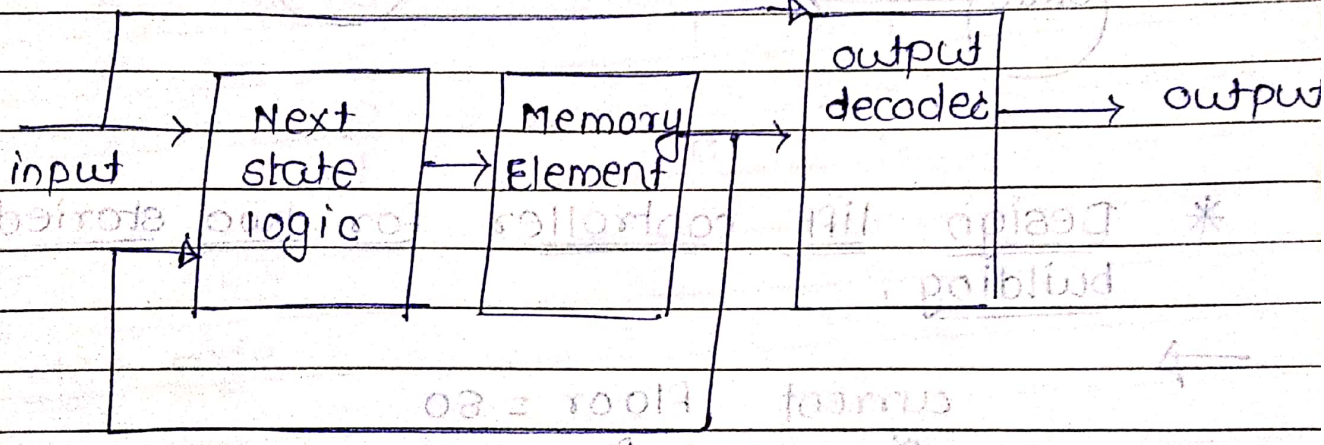
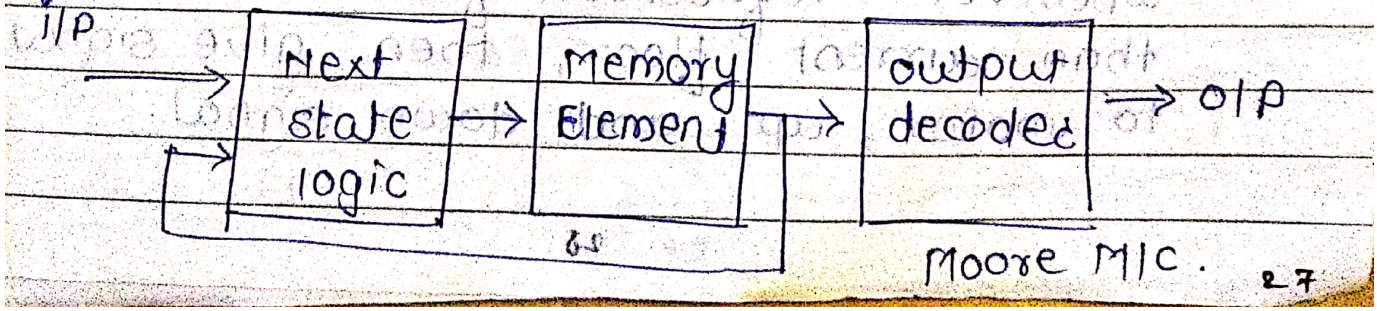


Fig. class A machine

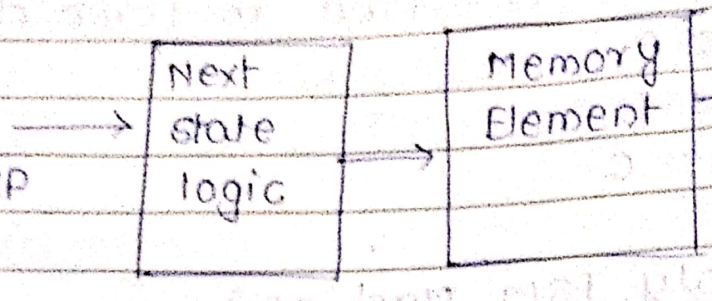
class B (Moore Machine)

In moore machine output is only functio
present state.

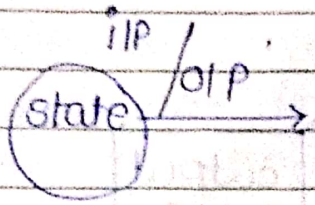


class c (Mealy and Moore M/c)

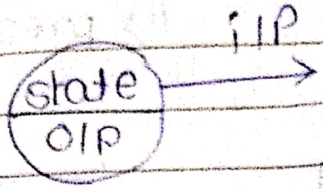
class c type is combination of the mealy and moore machine.



Mealy Machine state diagram



Moore Machine State diagram



* Design lift controller for two storied building.

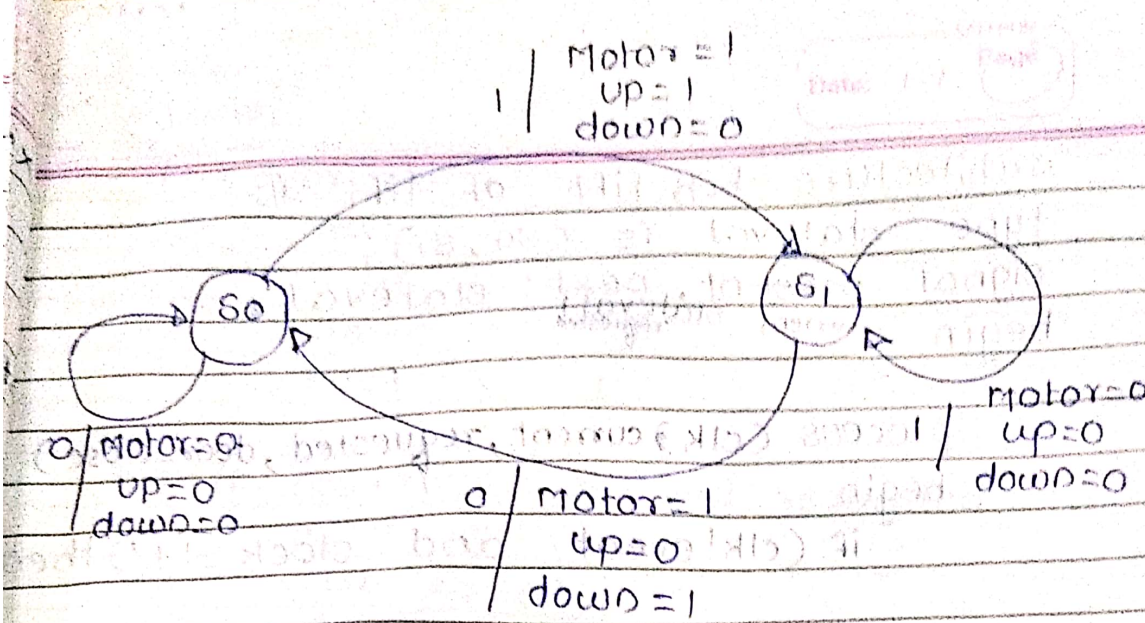
⇒

current floor = S0
Requested floor = S1

Assume mealy machine implementation.

When the lift door is open, buzzer switched on.

Whenever requested floor is other than current floor, then give signal to motor, up and down signal.



input | output

current Floor	Requested floor	Motor	up	down
0	0	0	0	0
0	1	1	0	0
1	0	1	0	1
1	1	0	0	0

VHDL code

```
entity lift is
    port (clk : in std_logic;
          current : in std_logic;
          requested : in std_logic;
          door_close : in std_logic;
          motor : out std_logic;
          up : out std_logic;
          down : out std_logic;
          buzzer : out std_logic);

end lift;
```

```
architecture beh_lift of lift is
type stateval is (s0, s1);
signal present, next : stateval ;
begin

process (clk) current, requested, door_clos
begin
    if (clk'event and clock = '1') t

case stateval is

when s0 =>

if (door_close = '0') then
        motor <= '0';
        up <= '0';
        down <= '0';
        buzzer <= '1';
else if (current = '0' and requested = '0')
        motor <= '0';
        up <= '0';
        down <= '0';
        buzzer <= '0';
else if (current = '0' and requested = '1') th
        motor <= '1';
        up <= '1';
        down <= '0';
        buzzer <= '0';
end if;
when s1 =>
```

```
if (door_close = '0') then
        motor <= '0';
        up <= '0';
        down <= '0';
        buzzer <= '1';

elseif (current = '0' and requested = '0') then
        motor <= '1';
        up <= '0';
        down <= '1';
        buzzer <= '0';

elseif (current = '1' and requested = '1') then
        motor <= '0';
        up <= '0';
        down <= '0';
        buzzer <= '0';
end if;
end case;
end process;
end beh_lift;
```

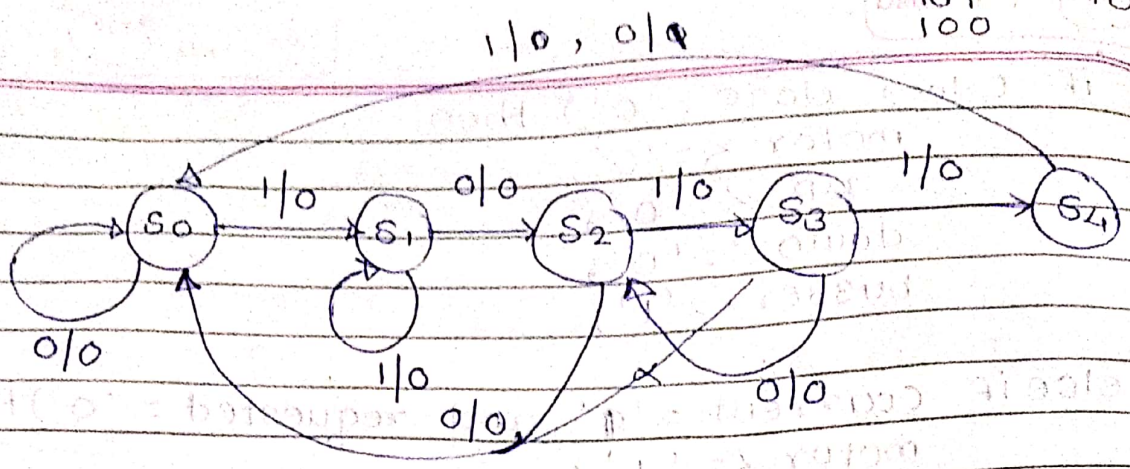
* Write sequence detector for 10110 using mealy and moore machine.

consider nonoverlapping sequence.

10110 10110 10110 10110 10110 ....

Mealy M/C :-

No of state = No of bits = 5



VHDL program

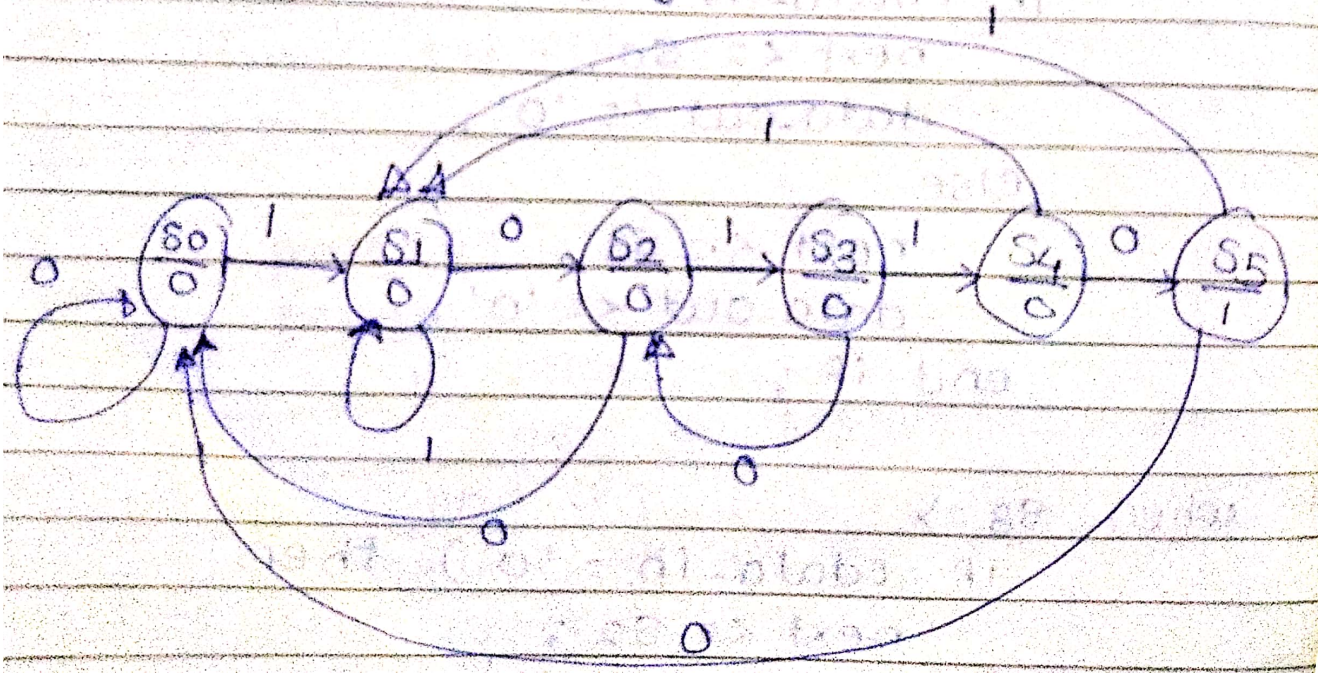
```
library ieee;
use ieee.std_logic_1164.all;

entity mealy1 is

    port( clock, reset : in std_logic;
          data_in : in std_logic;
          data_out : out std_logic);
end mealy1;

architecture beh_mealy of mealy1 is
type state is (s0, s1, s2, s3, s4);
signal present, next : state;
begin

    process (reset, clk)
    begin
        if (reset = '1') then
            next <= present;
        end if;
        else if ( clk='1' and clk'event)
    end process;
```

```
process (clk, data_in)
begin
    if (clk='1' and clk'event) then
      case present state is
when s0 =>
      if (data_in = '0') then
        next <= s0;
        data_out <= '0';
      else
        next <= s1;
        data_out <= '0';
      end if;

when s1 =>
    if (data_in = '0') then
      next <= s2;
      data_out <= '0';
    else
      next <= s1;
      data_out <= '0';
    end if;

when s2 =>
    if (data_in = '0') then
      next <= s0;
      data_out <= '0';
    else
      next <= s3;
      data_out <= '0';
    end if;

when s3 =>
      if (data_in = '0') then
        next <= s2;
        data_out <= '0';
```

```
else
    next <= S3;
    data_out <= '0';
end if;

when S4 =>
    if (data_in = '0') then
    next <= S0;
    data_out <= '0';
else
    next <= S0;
    data_out <= '0';
end if;

end case;
end process;
end beh_mealy;
```

Moore Machine:-

No of states = no of bits + 1
= 5 + 1
= 6



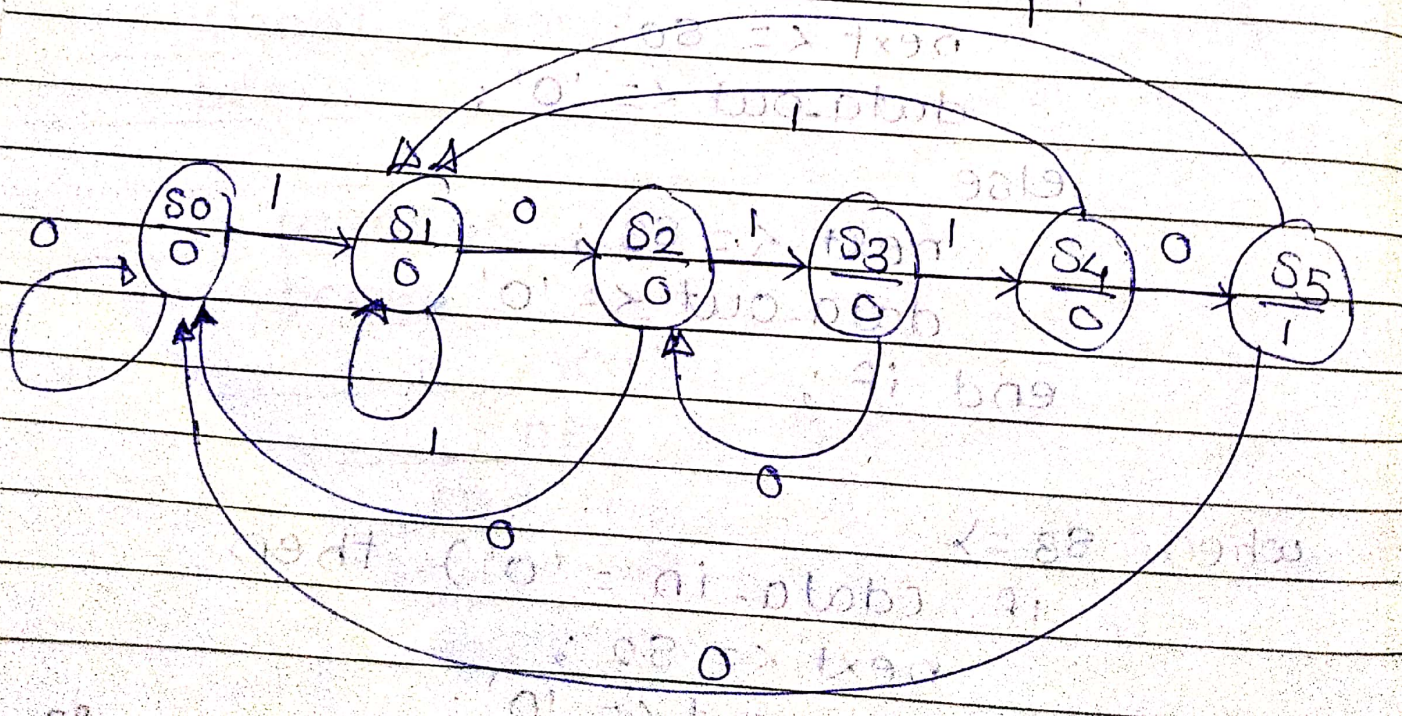
```
        else
            next <= S3;
            data_out <= '0';
        end if;

    when S4 =>
        if (data_in = '0') then
            next <= S0;
            data_out <= '0';
        else
            next <= S0;
            data_out <= '0';
        end if;

    end case;
    end process;
end beh_mealy;
```

Moore Machine:-

NO of states = no of bits + 1

= 5 + 1

= 6



```
library ieee;
use ieee.std_logic_1164.all;

entity moore1 is
  port
    ( clk, reset : in std_logic;
      data_in : in std_logic;
      data_out : out std_logic);
end moore1;

architecture beh_moore of moore1 is
type state is (s0, s1, s2, s3, s4, s5);
signal present, next : state;
begin

  process (reset)
  begin
    if (reset = '1') then
      next <= present;
    end if;
  end process;

  process (clk, data_in)
  begin
    if (clk'event and clk = '1') then
      case ~~state~~ present is
when s0=>
        if (datain = '0') then
          next <= s0;
        else
          next <= s1;
        end if;
        data_out <= '0';
```

```
when S1 =>
        if (data_in = '0') then
            next <= S1;
        else
            next <= S1;
        end if;
        data_out <= '0';
when S2 =>
        if (data_in = '0') then
            next <= S0;
        else
            next <= S3;
        end if;
        data_out <= '0';
when S3 =>
        if (data_in = '0') then
            next <= S2;
        else
            next <= S4;
        end if;
        data_out <= '0';
when S4 =>
        if (data_in = '0') then
            next <= S5;
        else
            next <= S1;
        end if;
        data_out <= '0';
when S5 =>
        if (data_in = '0') then
```

```
        next <= S0;
    else
        next <= S1;
    end if;
    data_out <= '1';

end case;
end process;
end beh_moore;
```

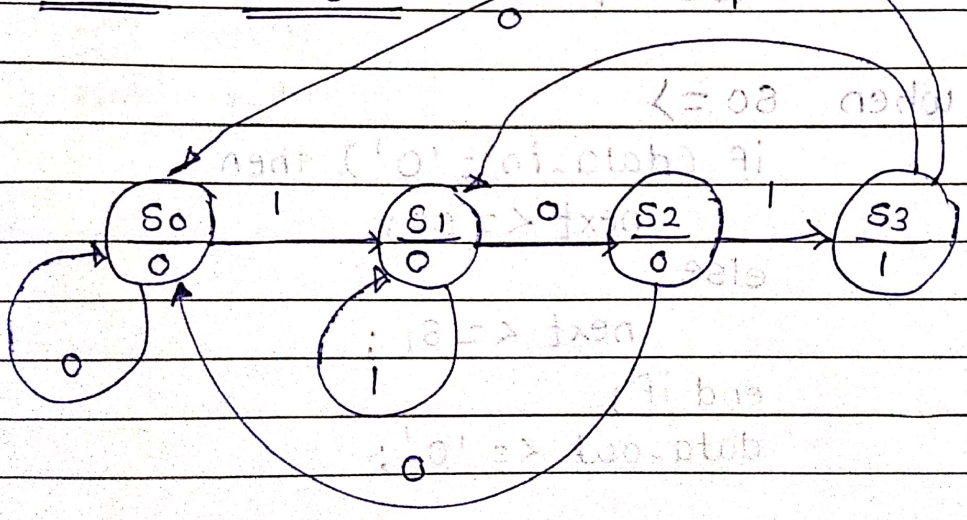
* Draw FSM and write VHDL code for 101 moore detector.

=> No of states = no of bits + 1

= 03 + 1

= 04

Moore detector




```
entity moore is
  port ( clk, reset : in std_logic;
         data_in : in std_logic;
         data_out : out std_logic);
end moore;
```

```
architecture beh.moore of moore is

type stateval is (s0, s1, s2, s3);
signal present, next : stateval;
begin

    process (reset)
    begin
        if (reset = '1') then
            next <= present;
        end if;
    end process;

process (clk, present)
begin
    if (clk'event and clk = '1') th
    case present is

when s0 =>
    if (data_in = '0') then
        next <= s0;
    else
        next <= s1;
    end if;
    data_out <= '0';

when s1 =>
    if (data_in = '0') then
        next <= s2
    else
        next <= s1;
    end if;
    data_out <= '0';
```

```
when S2 =>
    if (data_in = '0') then
        next <= S0;
else
        next <= S3;
end if;
        data_out <= '0';

when S3 =>
    If (data_in = '0') then
        next <= S0;
    else
        next <= S1;
    end if;
        data_out <= '1';
end case;
end if;
end process;
end beh_moore;
```