UNIT-1

→ Introduction to VHDL :

- VHDL stand for VHSIC H/w descriptive language
  ↳ (Very High speed integrated ckt)
- It is a language to describe the structure and behaviour of digital electronic H/w design.
- It is case insensitive
- Statements are terminated in VHDL with a semicolon.
- VHDL can be used to take 3 different approaches for describing H/w
  (i) Dataflow
  (ii) Behavioral
  (iii) Structural

→ Design Units :

There are 5 types of design units in VHDL which are
(i) Entity
(ii) Architecture Declaration
(iii) Configuration Declaration
(iv) Package declaration
(v) Package Body

(i) **Entity** : It is a list of I/O port (terminal) to the simple device or complex system. It also gives idea of external view.

Syntax :

```
Entity Entity_name is                    (for spacing)
port ( port name : signal_mode signal type)   (type → i/p, o/p, buffer)
                                              (type → bits, integers)
  :
End Entity - name
```

e.g

```
Entity AND-gate is
port ( A : in bit ;
       B : in bits ;
       Y : out bits ;)
End AND - gate
```

(ii) **Architecture** : It tell us about the internal structure and type of modelling

Syntax :

```
Architecture architecture_name of entity_name is
{declarations}                  ↳ (Type of modeling
                                   dataflow
                                   behavioral, structural)
 begin ;
 code
End architecturure_name
```

(iii) **Configuration**: It is used to associate an entity with an architecture. An entity can have more than one entity architecture associated with entity. The configuration assign a unique name for each architecture. The confi assign a unique name for each architecture pairing. Choose the config. name simply for the req. architecture.

Syntax:

```
configuration configuration_name of entity_name is
 [configuration declarations]
 for architecture_ name
  [ configuration item ]
 end for ;
end configuration configuration_name.
```

e.g

```
begin
A1 : ALU port map (...)
M1 : MUX port map (... )
M2 : mux port map(.. )
L1 : Latch port map (...)
L2 : Latch port map (...)
configuration cfg of microprocessor is
for structure                    (config. name)   (entity name)
  for A1 : ALU
  end for
for M1, M2; MUX
  use entity multiplexer
  end for
for worklatch
  for L1, L2 : Latch
  end for;
end for;
end configuration
```

(iv) **Package declaration**: A package is a collection of types, constants, subprog. usually intended to implement service or provides a mechanism to store items that can be shared across modules.

Package declaration contains a set of declarations which can be shared by design units, which are to be visible to users of the package can be accessed using LIBRARY and USE clause. (Var, sign etc which needs to be called n no. of times)

Syntax:

```
package (package name) is
component declaration
Signal declaration
constant declaration
variable declaration

end ( Package name)
```

(v) **Package Body**: It basically contains the behaviour of the subprograms in the package. An item declared in the package body can be used only within the body only.

```
package body package-name is
package-body-declaration
subprogram bodies/declaration
return package-name is
end package body package-name;
```

→ **Data Objects**:

A data object holds a value of specified type created by using object declaration. Every data object belongs to following 4 classes.

(i) constant
(ii) Variable
(iii) Signal
(iv) file

(i) **Constant**: An object of constant class can hold a single value of a given type and this value is assigned before simulation and can't be changed during simulation.

- needs to be assigned every time subprog is called.

Declaration:

```
constant Rise_time : Time = 10ns
constant Bus-width : Integer = 8;
```

`constant object : value type = value assigned.`

(A const. with no assigned value is called deferred const)

(ii) **Variable**: It also hold a single value of a given type

- different values can be assigned at diff times.

Declaration:

```
variable ctrl-status : bit vector (10 down to 0)
variable sum : integer range 0 to 100 := 10
```

(:= initial value)

`variable symbol : type [:= initial value]`

When simulation starts sum has intial value = 10

If no initial value is specified a default value is used as initial value.

(iii) **Signal**: Object belonging to this class holds a list of values (current value) and a set of possible future values.

Declaration:

```
signal CLOCK : BIT;
signal DATA-BUS : BIT_VECTOR(0 to 7)
```

(initial value of clk is 0)

`signal sig-name : data type [:= initial value]`

(iv) File: Objects belonging do this class contains seq. of values
- Values can be read/write to the file by read/write procedures

Declaration:

(ready text indefinite strings) (read only file)

type stimulus : Text open readmode is "/user/home/ade";

file vectors : bit_file is "/user/home/ add vec";

host environment (path of loc.)

(no mode means default read mode)

file file_name : file type_name [open mode] string expression;

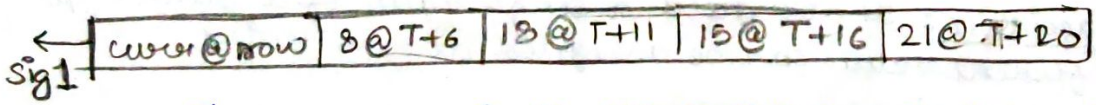
→ Signal Drivers:

A signal's value is a function of its current value and values of its drivers. Each process or assigned to the signal has corresponding driver value for that signal.

- Driver of a signal holds current as well as future values as a sequence, as each assignment identifies the value to appear of the signal.
- It is basically a que which indicates what values a signal gives at diff process assignments
- A driver contains at least one transaction value which could be initial value

e.g 4 waveform element and one driver is created for signal

```
process
begin
:
sig_1 <= 8 after 6ns, 18 after 11ns, 15 after 16 ns, 21 after 20ns
end process
```



The 1st transaction is the current value of signal. When stimulation starts and time advance to T+6ns the 1st transaction is deleted and sig gets value 8. When time advance to T+11ns sig gets value 18 and so on.

→ Delay:

The delay mechanism allows introducing propogation time of described system in signal assignment statements and are not allowed in variable assignment.

There are 2 type of delay models

(i) Inertial delay → used for modelling delay associated with switching ckts i.e for mod. noise

(ii) Transport delay → used for prop. of signal through lines or bus

(i) **Inertial Delay**: when it is used both signal value and delay value affect the deletion and addition of transactions

- If delay of new transaction < existing one. the latter is delet
- If delay of new transaction > existing one; signal values are compared.

  If same new transaction is added
  If not the existing one is deleted and new is added.

e.g data <= 9 after 15 ns, 19 after 25 ns, 27 after 17 ns.

curr@now	9@T+15ns

curr@now	19@T+25ns

as (9 ≠ 19) (25 > 15) → (case 2) as new sig is larger case 2.(ii)

curr@now	27@T+17ns

25 > 17 hence previous as it is

(ii) **Transport Delay**: either one pulse is genuine or noise it is passed as the output. Used for modeling propogation delays. Rules as above.

e.g data <= transport 19 after 15 ns, 12 after 28 ns, 27 after 18ns

curr@now	19@T+15ns	12@T+28ns

28ns > 15ns

curr@now	19@T+15ns	27@T+18ns

28ns > 18ns

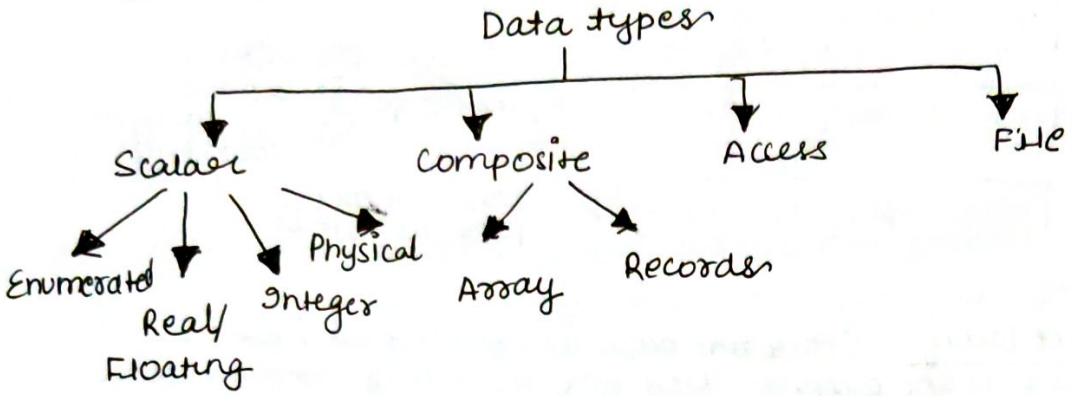
• Inertial delay is the default delay.	Transport delay is the wire delay
• used for switching circuits	used for modeling propogation
• It is the time taken by a signal to change its value	It is the time taken by a signal to travel across a wire.

(iii) **Delta Delay**: It is infinitely ~~not~~ small delay which is not real delay but used ~~for~~ in stimulation ~~to~~ in order to module the hardware concurrency accurately.

- used for update the value of a signal due to small delay.
- This delay models hardware where a minimal amount of time is required for a change to occur at stimulation time during stimulation.
- The event always occur at stimulation time + multiple of delta delay.
- Used for escaping unintended races

→ Data types:

- Signals have data types which limits the no. of possible values of that data.
- They can be predefined or user defined datatypes
- They needs to be fixed by signal is declared either as entity part or an internal architecture and can't be changed during run time.
- When signal values are updated data types on both sides of assignment operator should match.



(i) Signal Types: They have exactly one value and don't have distinguishable elements. Further classified as.

a) Integer: Range of integer value within a specified range All the predefined mathematical functions apply to this type
Min. range −2147483647 to +2147483647

e.g
```
process (x)
variable a: integer;
variable b : int_type;
begin
a := 1;        OK
a := -1;       OK
a := 1.0       error (decimal pt)
```
→ (considered as real no. value)

b) Real: used to represent no. out of the range of integer values as well as fractional values. Range from
−1.0E+38 to +1.0E+38

e.g
```
begin
a <= 1.0          OK
a <= 1            error
a <= 1.0E10       OK
a <= 1.5 E-20     OK
a <= 5.3 ns       error
```
→ (integer type)
→ integer

c) Enumerated: All the values of this type is user defined and represent the exact value req. for specific operation.
The values can be identifiers or literals

e.g enumerated type for 4 state simulation
Type bow is ('X', '0', '1', 'Z')

Each represent a unique state
'X' → Unknown value
'0' → A logic 0 or false value ← (character literals are needed for them as they are integer values. else error gen.)
'1' → A logic 1 or true value ←
'Z' → A tristate

d) Physical : used to represent physical quantities as distance, current, time etc. They act as a base unit for successive units.

e.g type current range 0 to 10000000 uni
na; → nano amps
ua = 1000na; → microamps
ma = 1000ua; → milli amps
a = 1000ma → amps
end units

(ii) Composite Types : They are classified as below:

a) Array : It groups one or more element of the same type as a single object by array indices

e.g type data_bus is array (0 to 31) of bit;

b) Record : VHDL provides basic facilities for record which are collection of elements of possibly diff. types

e.g Type V_I is
Record
N1 : Integer
N2 : Time
N3 : Real
End record;

[N1 assigned 22
N2 <= 10
N3 <= 1.0]

Variable V1 : V_I
V1 := (22, 10, 1.0)

(iii) Access Types : It is similar to a pointer in C language
Basically it points to the address of a specific object.
It can also have null value means it doesn't point to any address.
I predefined funct^n is available for access types that are NEW and ~~ALLOC~~ DEALLOCATE.

NEW → allocate memory of the size of the object and return the access value. In simple words new object is created by NEW.

DEALLOCATE → Takes the access value back and returns the mem. back to system or return the storage occupied by the system. and set it to NULL.

(iv) File Types: It provides access to other objects and used to read or store data in a file. It has 2 parts. File type declaration and File object declaration.

File type describes the base type of file eg integer_file.
File Object dec. describes name of file object, mode of file and physical path name. (IN or OUT).

e.g. FILE : integer_file IS IN "/doug/test/data.fu" (read) (write)

→ Concurrent Statements:
All the statements inside the architecture box are concurrent statements.

Concurrent	Sequential
(i) Can appear outside process	can't appear outside process block are inside process block
(ii) are inside arch. block	
(iii) Simple signal assignment statement	sequential variable assignment statement
(iv) No. of different processes may run at a time.	Process runs according to there sequence they are in
(v) Used in data flow	Behaviour
(vi) Event trigger	Not event trigger
(vii) executed at same time	one after another.
	Unconditional assig conditional selective ass

→ Subprograms:
They are seq. statements of procedure and functions. Both of them have body and may have declarations.

Procedure	Function
(i) It returns values in global objects. Return values in formal parameters	Return values as a value of the function.
(ii) ~~used in Behavioral modeling style~~	Do not change the formal parameter.
procedure proc_example (constant A: in integer; signal B: inout bit vector; var: C: out real; D: file)	function (sum (A: integer; B: integer) return integer
(iii) May contain return statement	surely have return statement
(iv) Return more than 1 value	Return single value
(v) used for processing	used for passing the data
(vi) parameters in mode in/out	mode in
(vii) executes in 0 time stimulator or not	operates in 0 time stimulation

→ Attributes:

It is a value, funct^n, type, range, const that can be associated with certain names within VHDL.

- They basically allow you to extract additional information about an object.
- They also allow you to assign additional info to objects in your design description.
- They can be user defined or predefined
- Helps to reduce the complexity of program.

Predefined attributes: There are 5 types of predef. attributes

(i) Value: Return const. value
(ii) Function: Call a function that return value
(iii) Signal: create a new signal
(iv) Type: Return a type name
(v) Range: Return a range.

User defined attributes: They are constant of any type. They are declared using attribute declaration which includes name and type of attribute

```
attribute attribute_name : value_type
```

→ Generics:

(i) It is used to transfer parameters → make code more flexible
(ii) Can be different data types
(iii) Declared in entity declaration
(iv) Can have a default value → (given in entity dec.)
(v) It basically allow you to create a flexible data structure that allow you to defere its data types at instantiation
(vi) declares const. object with in mode.

For a clearer view suppose if we wish to write on 2 RAM which are similar, same interface but diff access time. We don't need to write it twice one same module. By just parameterising the components during the instantiation with generic clause it can be done.

Just values needs to be customised are passed using generic clause.

Syntax:

```
generic ( generic_interface_list);
```

e.g

```
entity OR gate is
generic (N: natural)
port (A: IN bit vector 0 to N-1)
      Z: out bit vector)
end OR gate
arch. generic of OR gate is
begin
process (A)
begin
output := '1'
for k in 0 to N-1 loop
output := output and A[k]
exit when
output = '1'
end loop
Z <= output
end process
end generic.
```

Generate:

It is used to reuse or map a same statement in structural modeling. By generate statement we can map replica of any module.

(i) Concurrent statement & alternative to structural modelling
(ii) type of loop and used outside process
(iii) makes code compact

There are 2 types of generate statement

- For loop generate statement
- If loop generate statement

(i) For loop generate statement:

```
Generate label : For (var name) IN (range) generate
                 (entity name) : (comp. name)
end gen. statement label;
```

(no. of times loops applied)
(except pure numeric number)
(prog's entity)
(for which subprog or substatepart)

e.g entity and10 generate is

```
port (a,b: in STD_logic_vector (9 down to 0)
      c : out STD_logic vector (9 down to 0)
end and10 generate;

architecture behavirol of and10_generate is

component and123 is
port (a,b: in std_logic;
      c : out std_logic);
end component

begin
tab1 : for n IN 9 down to 0 generate and10_generate : and123
port map (a => a(n), b=>b(n), c=> c(n))
end generate lab;

end behavirol;
```

entity
component

(ii) If loop generate statement:

```
Generate
if (condition)
  // statement 1
else
  // statement 2
end Generate.
```

Syntax

e.g

```
parameter n=7;
          m=7,
          out=7;
generate
if (m<8) (n<8)
   and (r, p, q)
else
   or (r, p, q)
```

→ IEEE standard logic library :

The std_logic1164 is a package that defines a standard for designers to use in describing the interconnection data types used in VHDL.

The logic system in this package maybe insufficient for modelling switch transistors → (limitation)

→ File I/O :

File can be read as input and output by using pre-defined in and out std_logics.
They are mapped to the std-input & std-output for the host environment of the stimulator.

file INPUT : TEXT open READ_MODE is STD_INPUT;

→ Test Bench :

For checking a code stimulation is used but sometimes the code may be syntaxtically correct but not logically correct. In this case repeating the whole procedure again is tiresome hence for functional verification a test bench is written and by this test bench the we can verify or check the program.

- A HDL code is written to test another HDL module which is called Unit under test.
- The test bench method is not synthesisable, can only be used for stimulation.

A test bench is HDL code that provide documented, repeated set of stimuli that is portable across diff stimulators.

(i) Write a new module without any i/p or o/p signals
(ii) Declare all i/p of your UUT as reg and op as wire
(iii) Instantiate this with instance name.
(iv) Give stimulus to i/p.
(v) use # for delay in the i/p

e.g module test $y = (\bar{b}.\bar{c}) + (\bar{a}.\bar{b})$

```
sillyfunction ( input a,b,c,
                output y);
```

```
module testbench();
reg (a,b,c);        <-> i/p and o/p
wire y;
silly function uut (a(a), .b(b), .c(c), .y(y)):d
                                  -> (instantation)
initial begin
a=0, b=0, c=0; #10      <-> delay
a=1, b=1; c=0: #10
c=1; #10
a=1; b=0; c=0; #10
 end
end module
```

→ **Instantiation:**

It is a concurrent statement that can be used to connect circuit elements in a design.

It describes behavior of how components are connected.

It has 3 parts

(i) Label → identifies instance of component

(ii) Component type → select desired comp.

(iii) Port map → connect comp to sig.

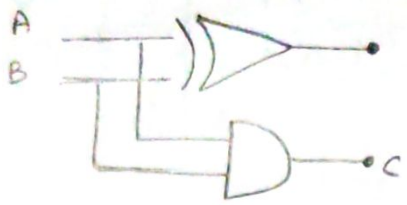
```
U1 = reg1 PORT MAP (d => d0, clk => clk, q => q0)
```

label ↙ ↳ component type

port map

UNIT-2

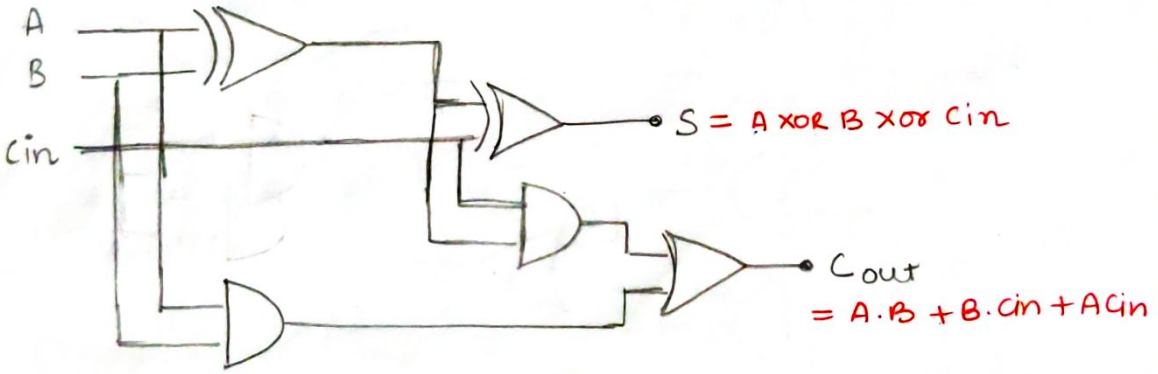
→ VHDL Code For Half Adder:



A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

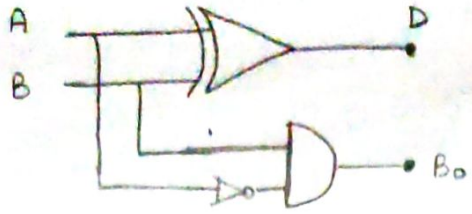
```
library ieee;
use ieee std_logic_1164.all
entity hA is
port (A, B : in bits
      S, C : out bits);

architecture hA of hadf adder
begin
  S <= A EXOR B
  C <= A AND B
  end hA.
```

→ VHDL Code For Full Adder:




```
library ieee;
use ieee std_logic_1164.all
 entity FAI
 port (A, B, Cin : in bits;
       S, Cout : out bits);
 end FAI

 architecture of FAI of full adder is
 S <= AXOR B EXOR C
 Cout <= A AND B OR B AND Cin OR A AND Cin
 end FAI.
```

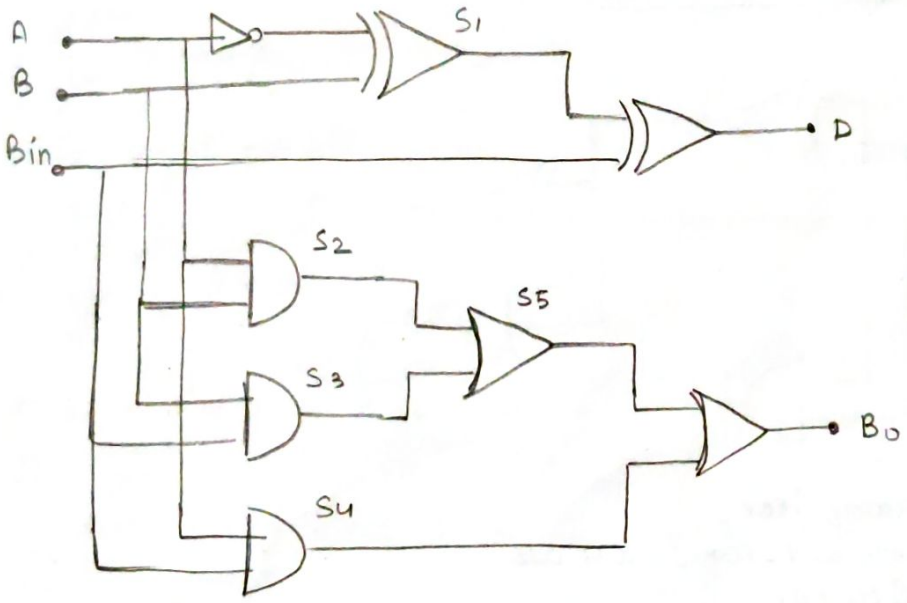
→ Half Subtractor :



A	B	D_0	B_0
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

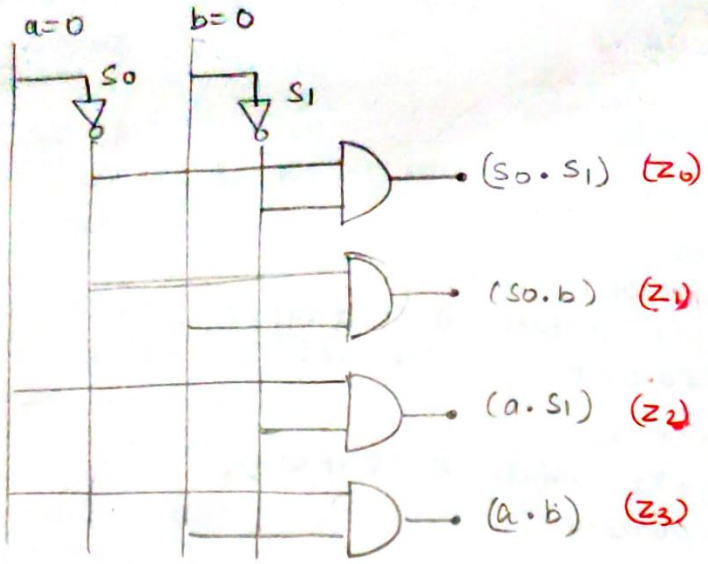
```
library ieee;
use ieee.std_logic_1164;
entity HS1
port ( A, B : in bits
       D, B0 : out bits);
end HS1
architecture of HS1 of half subtractor is
 D <= A XOR B
 B0 <= (NOT A) AND B
 end HS1;
```

→ Full Subtractor :



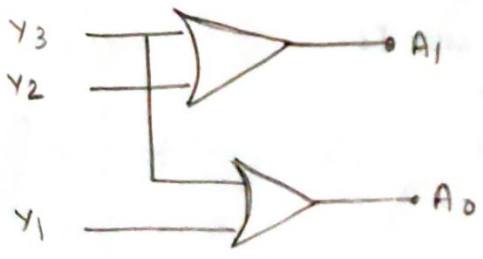

```
library ieee;
use ieee.std_logic_1164.all
 entity FS1
port ( A, B, Bin : in bit
     D, B0 : out bit)
 end FS1
architecture of HS1 of full adder is
 D <= (NOT A) XOR B XOR Bin
 B0 <= (NOT A) AND B OR B AND Bin OR Bin AND (NOT A)
 end FS1
```

Decoder :




```
library ieee;
use ieee.std_logic_1164.all
entity Dec
port (a, b : in bits
      S0, S1, Z0, Z1, Z2, Z3 : out bits);
end dec.
architecture of dec is
begin;
 S0 <= NOT a
 S1 <= NOT b
 Z0 <= S0 AND S1
 Z1 <= S0 AND b
 Z2 <= a AND S1
 Z3 <= a AND b
 end dec;
```

Encoder :



Y4	Y3	Y2	Y1	A1	A0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	⊗	⊗	⊗	1	1

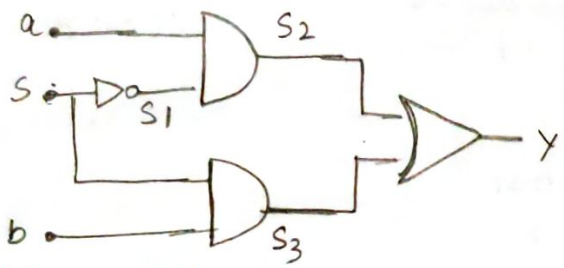
```
library ieee;
use ieee.std_logic_1164.all
entity Enc
port (Y3, Y2, Y1 : in bits
      A1, A0 : out bits)
end Enc.
```

```
architecture abc of Encoder is
A1 <= Y3 OR Y2
A0 <= Y3 OR Y1
end abc.
```

Structural modeling:

```
entity Enc
component OR1
port ( Y3, Y2 : inbits ; A1 : out bit);
end component

component OR2
port (Y3, Y1 : inbits ; A0 : out bits);
end component

begin;
L1 : OR1 portmap( Y3, Y2, A1)
L2 : OR2 portmap ( Y3, Y1, A0)
end Enc.
```

→ Multiplexer:



S	Y
0	a
1	b

00	a
01	b
10	c
11	d

Data flow modeling:

```
library ieee;
use ieee. std_logic_1164. all
entity MUX
port (a, b : in bits; Y: out bits);
end MUX

architecture abc of MUX is
 Y <= a  when S=0
 Y <= b  when S=1
 end abc.
```

Structural Modeling:

```
entity MUX
component NOT1
port (S: in bit ; S1 : out bit)
end component.
```

```
component AND1
port (a, S1 : in bit ; S2 : out bit)
end component

component AND2
port (b, S : in bit ; S3 : out bit)
end component

component OR1
PORT (S2, S3 : in bits ; Y : out bit)
end component

Port (S2 :
architecture of abc
L1 : NOT 1 (S, S1)
L2 : AND1 ( a, S1, S2)
L3 : AND2 ( b, S1, S3)
L4 : OR1 ( S2, S3, Y)
end architecture
```

Behavioral modeling :

```
begin
process (S, Y)
begin;
  if S <= 0 then
      Y <= a
  else
      Y <= b
  end if
end process
```

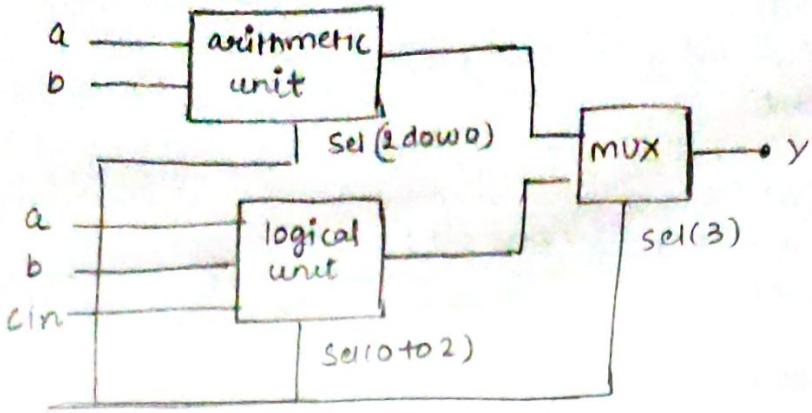
→ ALU :

Arithmetic

a	b	Sel	
0	0	0	a
0	0	1	a+1
0	1	0	a-1
0	1	1	b
1	0	0	b+1
1	0	1	b-1
1	1	0	a+b
1	1	1	a+b+Cin

Logic

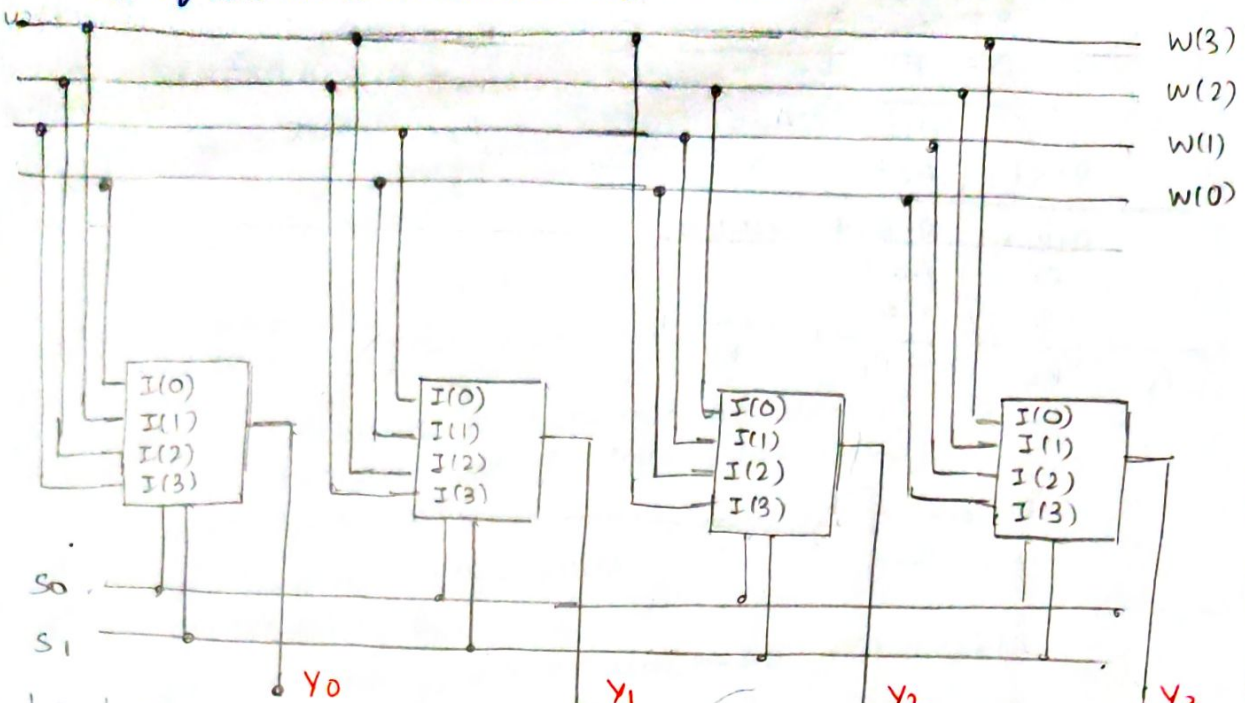
a	b	Cin	
0	0	0	Not a
0	0	1	Not b
0	1	0	A AND B
0	1	1	A OR B
1	0	0	A NAND B
1	0	1	A NOR B
1	1	0	A EXOR B
1	1	1	NOR (A XOR B)




```
library ieee;
use ieee.std_logic_1164.all
entity ALU
port (a, b, cin : in bit_vector (0 to 7)
      selc : in bit_vector (0 to 3)
      cin : in bit
      y : out bit)
end ALU

architecture abc of ALU is
begin
sel (2 down 0)
arithmetic <= a when '000'
           <= a+1 when '001'
           <= a-1 when '010'
            .
            .
            .
           <= a+b+cin when '111'

sel (2 down 0)
logic <= NOT a when '000'
      <= NOT b when '001'
      <= A AND b when '010'
       .
       .
       .
      <= NOT (A XOR B) when '111'

sel (3)
y <= arithmetic when '0'
  <= logical when '1'
end abc
```

→ Barrel shifter:

A barrel shifter shifts the input data by the specified no. of bits in a combinational manner.



S_0	S_1	O/P	Y_0	Y_1	Y_2	Y_3
0	0	I_0	W(0)	W(1)	W(2)	W(3)
0	1	I_1	W(3)	(W(0)	W(1)	W(2)
1	0	I_2	W(2)	W(3)	W(0)	W(1)
1	1	I_3	W(1)	W(2)	W(3)	W(0)

(lines selected)

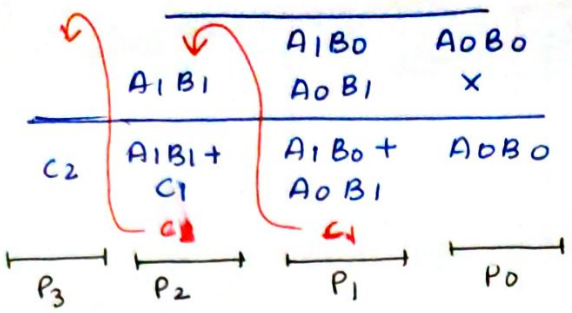
```
library ieee;
use ieee.std_logic_1164.all
use ieee.numeric_std.all;
 entity barrel4.all is
port ( W: in unsigned (3 down to 0)
       S: in unsigned (1 down to 0)
       Y: out unsigned (3 down to 0)
 end barrel4.

architecture abc of barrel 4 is
 begin
process (S, W)
  begin
 when '00' then
       Y <= W
 when '01' then
       Y <= WROR1;   → (rotate W one time)
 when '10' then
       Y <= WROR2;
 when '11' then
       Y <= WROR3;
end behavioral.
```

→ Multiplier :

For e.g 2 bit multiplier we have 2 no .

$A = A_1\ A_0$

$B = B_1\ B_0$

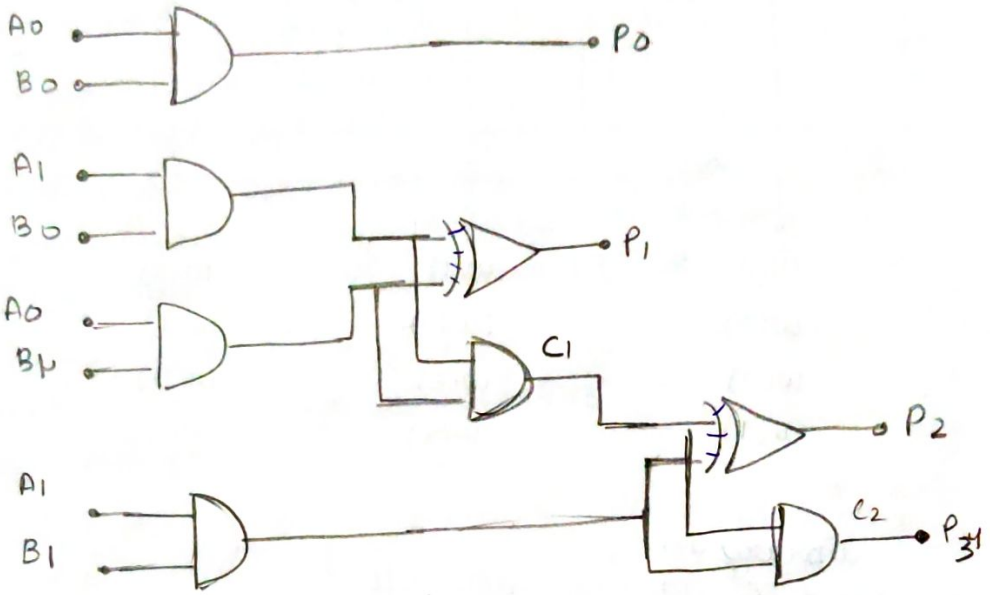


$P_0 = A_0 B_0$

$P_1 = A_1 B_0 + A_0 B_1$ } (Half adder is needed)

$P_2 = A_1 B_1 + C_1$ }

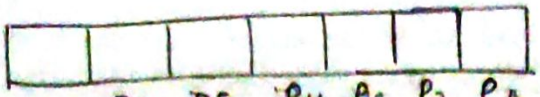
$P_3 = C_2$




```
library ieee.
use ieee. std_logic_1164. all
entity multiplier
port ( A0, A1, B0, B1 : in bits
       P0, P1, P2, P3 : out bits)
end multiplier
architecture abc of multiplier is
  P0 <= A0 AND B0
  P1 <= A1 AND B0 OR A0 AND B1
  P2 <= A1 AND B1 OR C1
  P3 <= C2
end abc
```

→ Hamming Code Encoder:

P → Parity bits
D → Data bits



D7 D6 D5 P4 P3 P2 P1
(D3) (D2) (D1) (P2) (D0) (P1) (P0)

$P_0 = D_3 D_5 D_7$

$P_1 = D_3 D_6 D_7$

$P_2 = D_5 D_6 D_7$

```
library ieee;
use ieee.std_logic_1164.all
entity HAM
port (datain : in bit_vector (3 down to 0)
      hamout : out bit_vector (7 down to 0)
end HAM
architecture abc of HAM is
signal P0, P1, P3 : bits
P0 <= (datain(3) XOR datain(1) XOR datain(2))
P2 <= (datain(0) XOR datain(2) XOR datain(3))
P3 <= (datain(1) XOR datain(2) XOR datain(3));
  hamout(4 to 6) <= P0, P1, P2
  hamout(0 to 3) <= datain(0 to 3)
  end abc.
```

UNIT-3

→ Synchronous And Asynchronous Circuits:

Synchronous	Asynchronous
(i) They are easy to design	Comparatively difficult to design
(ii) Clocked flip flop is used as memory element	Time delay or unclocked flip flop is used as timing element.
(iii) They are slower	They are faster
(iv) Edge triggering is there	Level triggering is there

→ Finite State Machine:

o/p depends upon PS and

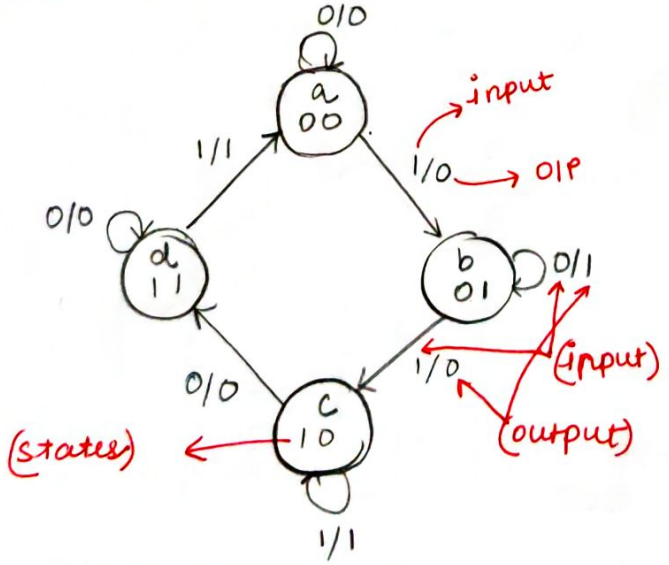
FSM is an abstract model describing the synchronous sequential machine and it spatial counter part, the iterative network. It is the basis of understanding and development of various computations.

OR

A state machine is any device that stores the status of something at a given time and can operate on input to change the status / cause.

→ State Diagram:

A state diagram is a pictorial representation of the relationship between the PS, i/p, NS and o/p of a sequential circuit.



PS	N.S		o/p	
	x=0	x=1	x=0	x=1
a	a	b	0	0
b	b	c	1	0
c	d	c	0	1
d	d	a	0	1

aa → 01

→ State Table:

The tabular representation of a state diagram is called a state table.

→ State Reduction:

This technique basically avoids the introduction of redundant states. This reduction reduces the no. of flipflops and logic gates reducing the cost and complexity of the circuit.

When 2 states are equivalent one of them can be removed without altering the i/o relationship.

Steps:

(i) Determine the state table for given state diagram
(ii) Find equivalent states.

e.g

P.S	N.S		O/P	
	X=0	X=1	X=0	X=1
a	a	b	0	0
b	c	d	0	0
c	d	d	0	0
d	e	f	0	1
e	a	f	0	1
f	g	f	0	1
g	a	f	0	1

cancel one of the two set

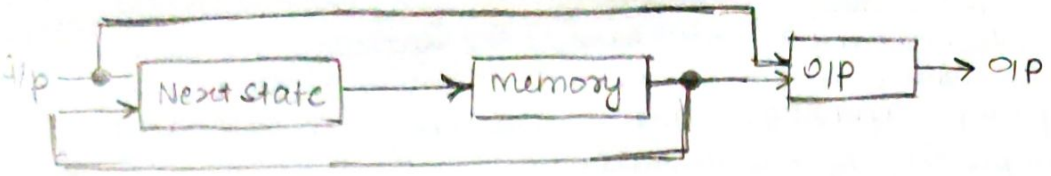
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	e	f	0	1

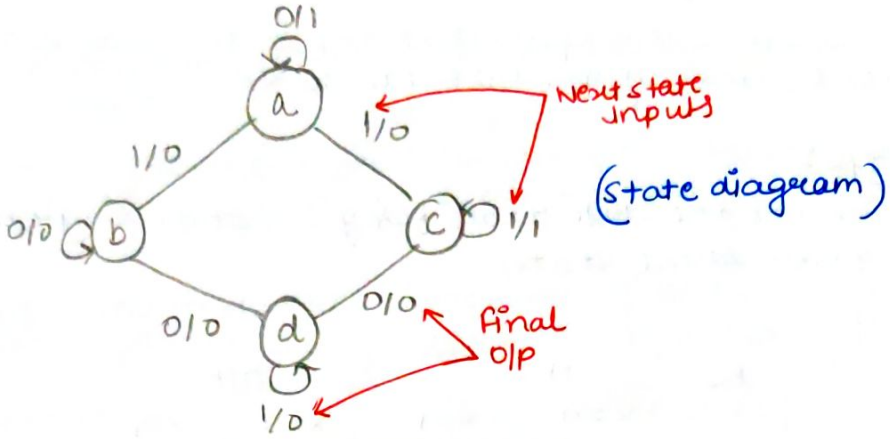
cancel one of them again

a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	d	0	1
e	a	a	0	1

→ Mealy And Moore:

(i) **Mealy:** In the mealy machine the output is a function of both the present states and the input.

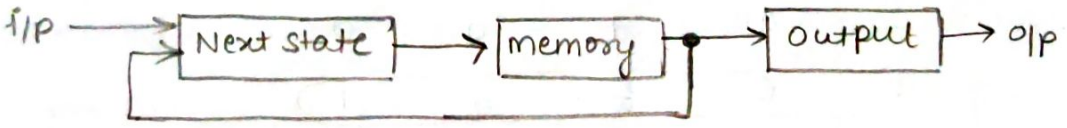


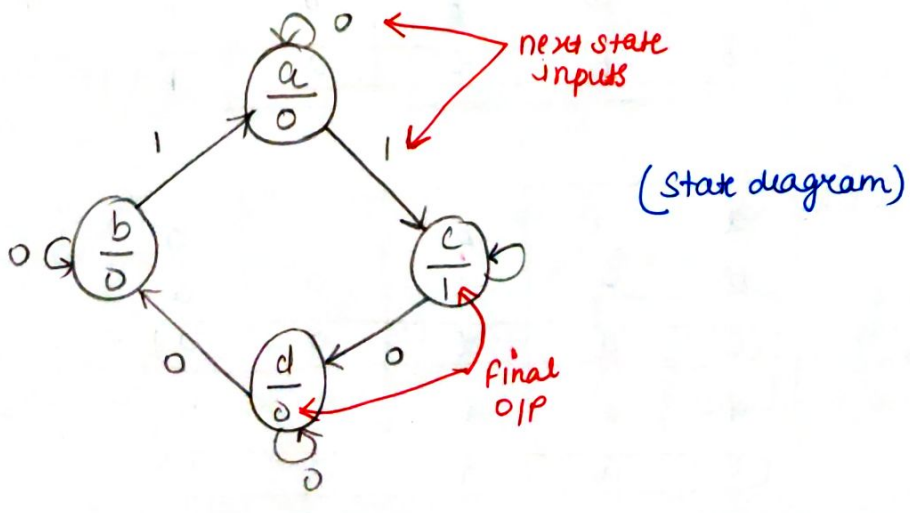


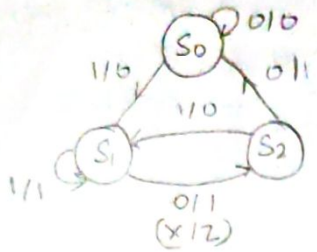
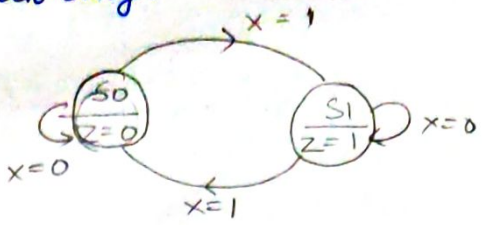
P.S	N.S		O/P	
	X=0	X=1	X=0	X=1
a	a	c	0	0
b	b	a	0	1
c	d	c	0	1
d	b	d	0	0

(state table)

(ii) **Moore:** In the moore machine the o/p is a function of the present state only.





	Mealy circuit	Moore circuit
(i)	o/p is a function of present state and input	o/p is a function of present state only.
(ii)		
(iii)	one state may have more than one output	one state have only one output
(iv)	o/p changes when i/p changes of there is change in FF's state	o/p changes only when one state changes
(v)	less no. of states required for implementation	more no. of states are required for implementation.
(vi)	more flexible	less flexible
(vii)	False output can appear when state has changed but i/p has not yet.	No false output may appear as o/p is not a function of i/p
(viii)	Outputs are written along with states	Different column is required

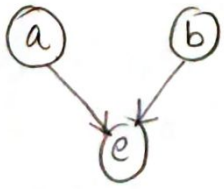
→ **State Assignments :**

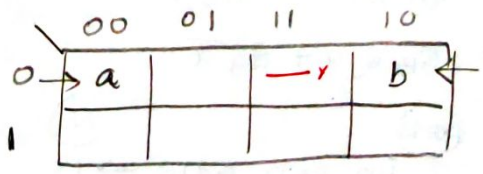
The process of assigning binary values to the states of sequential machine is known as state assignment. The values are assigned in such a way that it is possible to implement them using min logic gates.

As to minimise the hardware there are 2 rules to be followed

(i) States having same NS should be assigned binary values such that both the states are adjacent in K map.

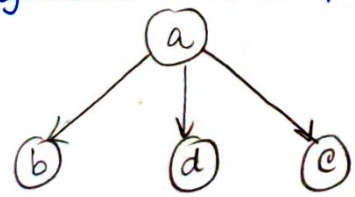
e.g.

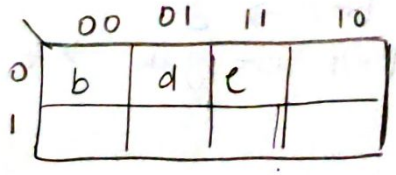




assign b in 10 or 01 to make in adjacent

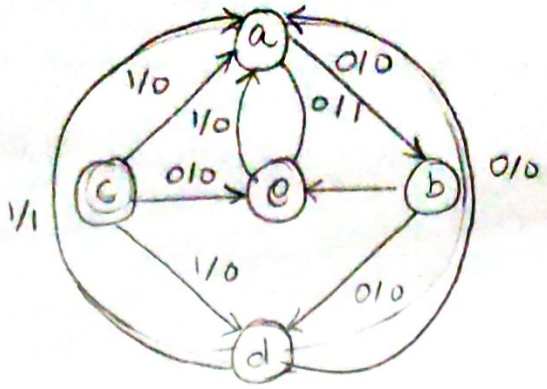
(ii) ~~Similarly~~ in rule 2 if PS is common the NS should be adjacent in K map





assign the NS adjacently and rest don't care

e.g



a = 000
b = 001
c = 010
d = 011
e = 100

P.S			i/p	N.S			o/p	
Q_A	Q_B	Q_C		Q_A^+	Q_B^+	Q_C^+		
0	0	0	0	0	0	1	0	a
0	0	0	1	0	1	0	0	
0	0	1	0	0	1	1	0	b
0	0	1	1	ϕ	0	0	0	
0	1	0	0	1	0	0	0	c
0	1	0	1	0	1	1	0	
0	1	1	0	0	0	0	0	d
0	1	1	1	0	0	0	1	
1	0	0	0	0	0	0	1	c
1	0	0	1	0	0	0	0	

Rest all states from (101 to 111) (10-15) don't care
As we are using D flip flop.

Characteristic eq ⇒ $Q_A^+ = D_A$

Sove for D_A^+, D_B^+, D_C^+ by K map of (16) result

$D_A = Q_B Q_C \bar{x} + Q_B Q_C x$

$D_B = \bar{Q}_A \bar{Q}_C x + \bar{Q}_B Q_C \bar{x}$

$D_C = \bar{Q}_A \bar{Q}_B \bar{x} + Q_B \bar{Q}_C x$

$Z = Q_B Q_C x + Q_a \bar{x}$

Ports	No
2 i/p AND gate	→ 1
3 i/p AND gate	→ 4
2 i/p OR gate	→ 4
FF	→ 3
1 i/p not gate	→ 4

Now with state assignment acc to the diagram

b and c should be adjacent
d and e should be adjacent

	00	01	11	10
0	a	b	c	
1		d	e	

Acc to the assignment

a = 000
b = 001
c = 011
d = 101
e = 111

P.S			I/P	N.S			O/P	
Q_A	Q_B	Q_C		Q_A^+	Q_B^+	Q_C^+		
0	0	0	0	0	0	1	0	a
0	0	0	1	0	1	1	0	
0	0	1	0	1	0	1	0	b
0	0	1	1	1	1	1	0	
0	1	0	0	x			x	
0	1	0	1	x			x	
0	1	1	0	1	1	1	0	c
0	1	1	1	1	0	1	0	
1	0	0	0	x			x	
1	0	0	1	x			x	
1	0	1	0	0	0	0	0	d
1	0	1	1	0	0	0	1	
1	1	0	0	x			x	
1	1	0	1	x			x	
1	1	1	0	0	0	0	1	e
1	1	1	1	0	0	0	0	

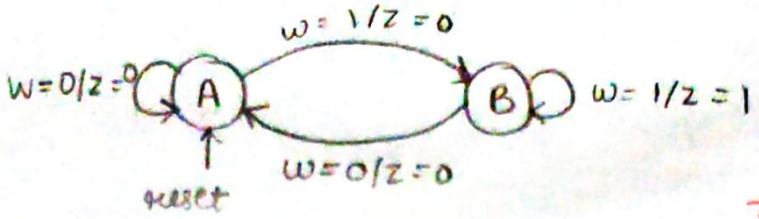
$D_A = \overline{Q}_A Q_C$

$D_B = \overline{Q}_A \overline{Q}_B x + \overline{Q}_A Q_B \overline{x}$

$D_C = \overline{Q}_A$

$Z = Q_A Q_B \overline{x} + Q_A \overline{Q}_B x$

→ VHDL code for Mealy Machine:



P.S	N.S		Z
	X=0	X=1	X=0
A	A	B	0
B	A	B	0

```
entity mealy is
port ( clock, reset, w : in std_logic ;
                     z : out std_logic );
end mealy
architecture behavioural of mealy is
type State_type is (A, B);
 signal y : State_type;
 begin:
process ( reset, clock)
 begin
if reset = '0' then
    y <= A
  else if ( clk = '1' then)
 when A =>
  if ( w='0' ) then y<=A
  else
     y <= B
 when B =>
  if ( w = '0' then y <=A
  else
   y <= B ;
  end if ;
  end case ;
  end if ;
  end process
 Process ( y, w)
  begin
  case y is
  when A =>
   z <= '0'
  when B =>
   z <= w ;
  end case ;
  end process;
  end behavioral ;
```

P.S	N.S		O/P	
	X=0	X=1	X=0	X=1
A	A	B	0	0
B	A	B	0	1

Ques: Code for the eqn using process statement

I/p: x, clk

O/p: $Z = x'Q_3' + xQ_3$

$Q_1^+ = Q_2^+$; $Q_2^+ = Q_1$, $Q_3^+ = Q_1Q_2Q_3 + x'Q_1Q_3' + xQ_1'Q_2'$

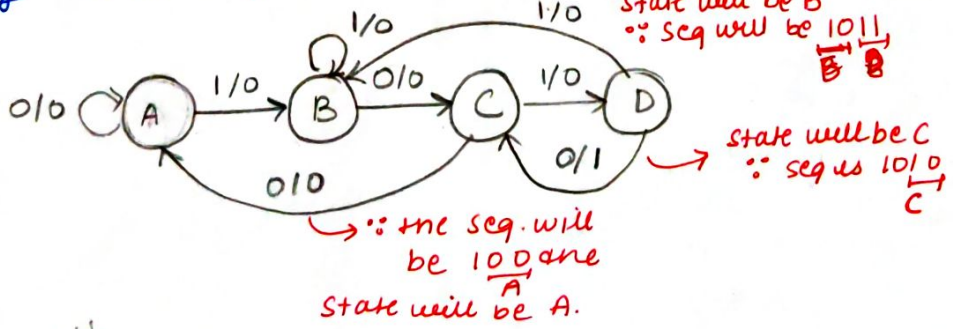
```
entity example is
port (x, clk : in std_logic;
      Q1, Q2, Q3 : inout std_logic
      Z : out std_logic);
end example;
architecture dataflow of example is
begin
process (clk)
begin
if (clk = '1' and clk event) then
 Q1 <= not Q2
 Q2 <= Q1
 Q3 <= (Q1 not Q2 not Q3) OR (not x and Q1 and (not Q3) or
       (x and (not Q1) and not Q2));
 Z <= ((not x) and (not Q3) or (x and (not Q3));
end if;
end process;
end dataflow;
```

→ **Sequence detector:**

A sequence detector is a sequential state machine which takes input string of bits and generates an output 1 whenever the required sequence is detected.

e.g. 10 10 sequence detector

A = 00
B = 01
C = 10
D = 11



P.S	N.S	
	X = 0	X = 1
A	A, 0	B, 0
B	C, 0	B, 0
C	A, 0	D, 0
D	C, 1	B, 0

(State table)

make excitation table and transition table.

Code:

```
library ieee;
use ieee.std_logic_1164.all
use ieee.std_logic_arith.all
entity sequencedet is
port( x: in std_logic;
      rest: in std_logic;
      y: out std_logic);
end sequencedet;
architecture behavior of seqdet is
type state is (A, B, C, D);
signal ps, ns: state;
begin;
process (reset)
begin
if rst = '1' then
  PS <= A
  else
  PS <= ns
end if
end process

process (PS, x)
begin
case PS is
when A =>
if x='0' then
ns <= A; y <= '0'
else if x='1' then
ns <= B, y <= 0
end if;

when B =>
if x='0' then
ns <= C; y <='0'
else if x='1' then
ns <= B ; y <=0
else if;

when C =>
if x ='0' then
ns <=  ; y <=0
.
.
.
end if
end process;
```

Mealy Model :

→ For Non overlap, move last bit to reset
→ For 1 bit Non overlap, compare the last bit to 1 bit state
→ For 2 bit Non overlap compare last 2 bits to 2 bit state if not similar then to 1 bit state and so on

Moore Model :

→ For NO, compare last bit to 1 bit state
→ For 1 bit NOL compare last 2 bit to 2 bit states, then 1 bit and so on
→ For 2 bit NOL compare the last 3 bit to 3 bit state, 2 bit and so on