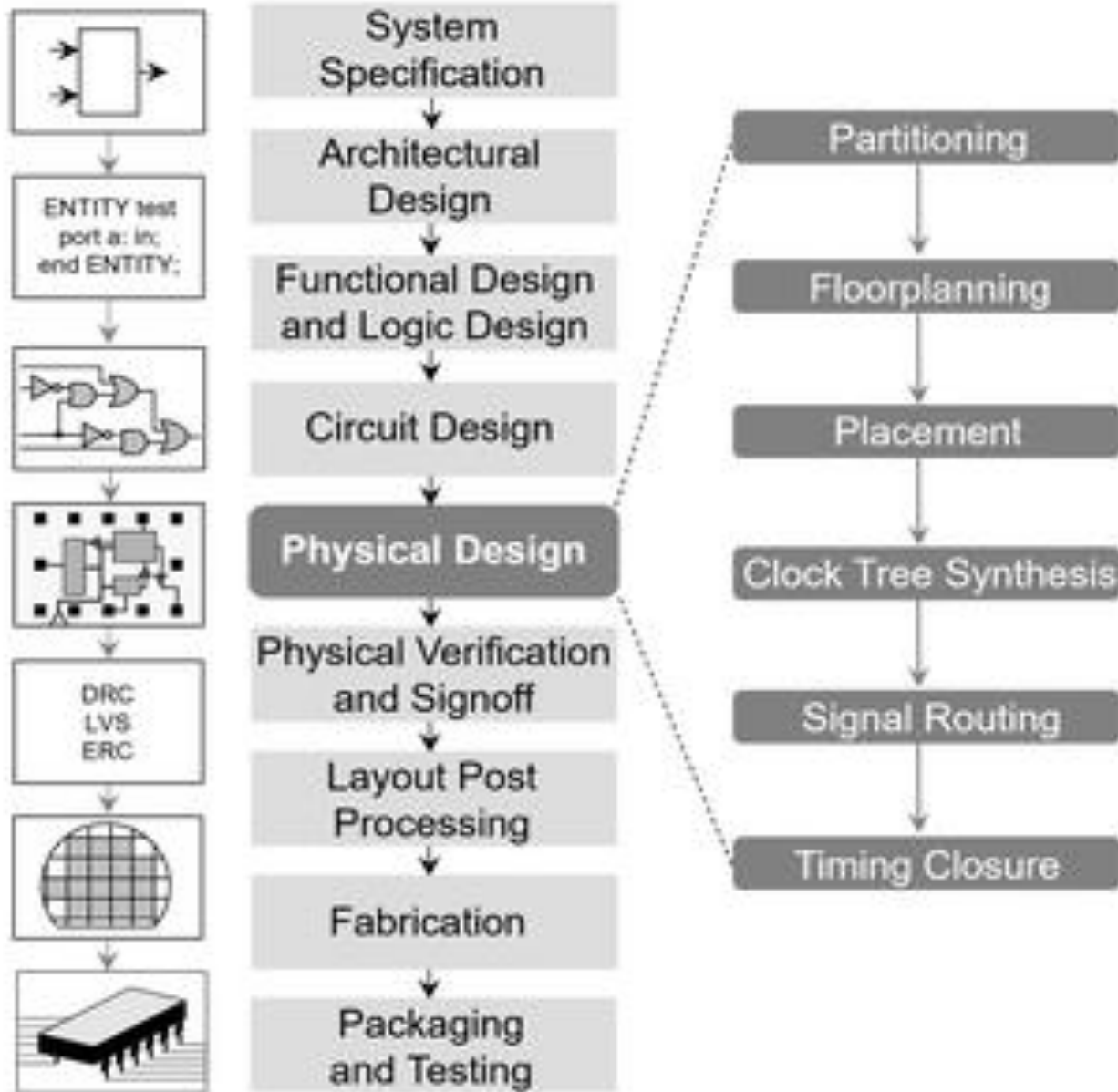


Advanced Digital System Design-

UNIT-II

VLSI Design Flow



What is VHDL?

- VHDL is the VHSIC (Very High Speed Integrated Circuit) Hardware Description Language

- A Simulation Modeling Language

- A Design Entry Language

- A Standard Language

- A Netlist Language

History of VHDL

- * 1981: Initiated in 1981 by US DoD to address the hardware life-cycle crisis**
- * 1983-85: Development of baseline language by Intermetrics, IBM and TI**
- * 1986: All rights transferred to IEEE**
- * 1987: Publication of IEEE Standard**
- * 1987: Mil Std 454 requires comprehensive VHDL descriptions to be delivered with ASICs**
- * 1994: Revised standard (named VHDL 1076-1993)**

Basic Terminology

- ❑ VHDL can be used to model a digital system.
- ❑ An entity is used to describe a hardware module
- ❑ VHDL provides five different types of primary constructs called design units. They are
 - i. Entity Declaration
 - ii. Architecture body
 - iii. Configuration declaration
 - iv. Package declaration
 - v. Package body

Entity declaration

It defines the names, input output signals and modes of a hardware module.

Syntax:

```
entity entity_name is  
    Port declaration;  
end entity_name;
```

- starts with 'entity' and ends with 'end' keywords.
- Ports are interfaces through which an entity can communicate with its environment
- Each port must have a name, direction and a type. The direction will be input, output or inout.

In	Port can be read
Out	Port can be written
Inout	Port can be read and written
Buffer	Port can be read and written, it can have only one source.

Architecture

- Describes the internal description of design or it tells what is there inside design.
- Each entity has at least one architecture and an entity can have many architecture.
- Architecture can be described using structural, dataflow, behavioral or mixed style.

Syntax:

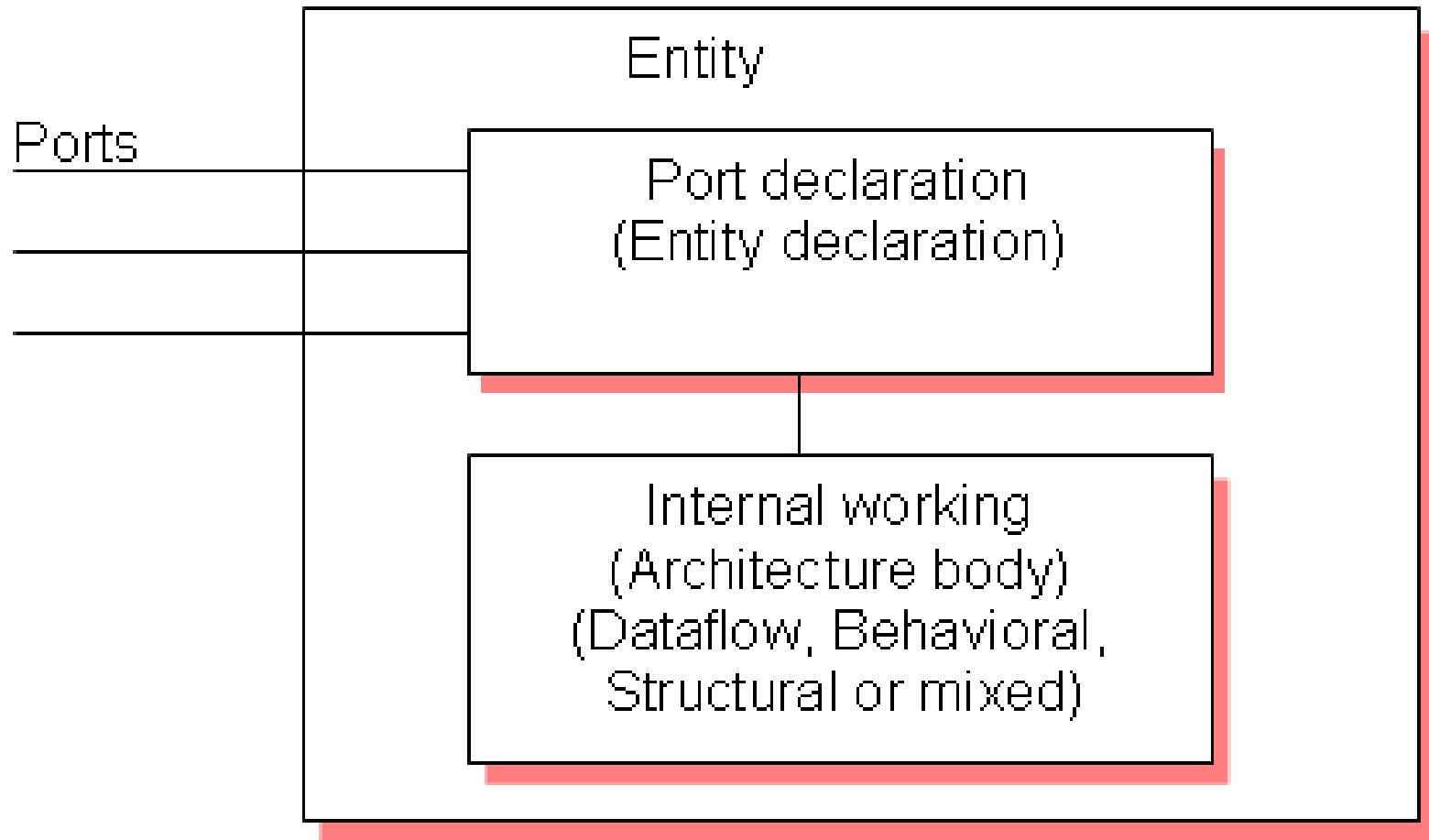
```
architecture architecture_name of entity_name  
    architecture_declarative_part;  
begin  
    Statements;  
end architecture_name;
```

Modeling Styles

The internal working of an entity can be defined using different modeling styles inside architecture body. They are

- Dataflow modeling.
- Behavioral modeling.
- Structural modeling.

Structure of an Entity



Basic Language Elements

Identifiers

- Set of upper case letters(A-Z), lower case letters(a-z), digit(0-9) and an underscore (_) can be used.
- First character must be a letter and last character may not be an underscore.
- **VHDL is not case sensitive.**
- **Comment line must be preceded by two consecutive hyphens(--)**

VHDL Operators

- Logical Operators: and, or, not, nand, nor, xor, xnor
- Relational Operators: = , /=, <, <=, >, >=
- Shift Operators: sll, srl, sla, sra, rol,ror
- Adding Operators: +, - , &
- Multiplying Operators: *, /, mod, rem
- Misc. :**,abs

Highest	Order of Precedence for Operators				Lowest
Misc.	Multiplying	Adding	Shift	Relational	Logic

- Shift Operators sll shift left logical (fill value is '0')
- srl shift right logical (fill value is '0')
- sla shift left arithmetic (**fill value is right-hand bit**)
- sra shift right arithmetic (**fill value is left-hand bit**)
- rol rotate left
- ror rotate right

all operators have two operands:

left operand is bit_vector to shift/rotate

right operand is integer for # shifts/rotates

- integer same as opposite operator with + integer

examples:

"1100" sll 1 yields "1000"

"1100" srl 2 yields "0011" **1101 sla 1011**

"1100" sla 1 yields "1000"

"1100" sra 2 yields "1111"

"1100" rol 1 yields "1001"

"1100" ror 2 yields "0011"

"1100" ror -1 same as "1100" rol 1

Multiplying Operation

mod & rem operate on integers & result is integer

rem has sign of 1st operand :

mod has sign of 2nd operand :

examples:

7 mod 4 -- has value 3

7 mod (-4) -- has value -3

& concatenation puts two bits or bit_vectors into a bit_vector

example:

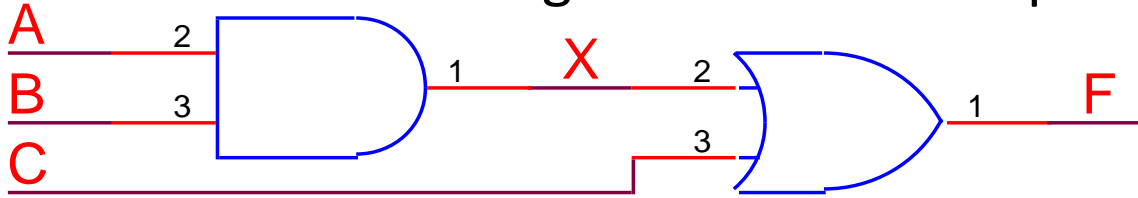
a=1 b=1

B <= '0' & '1' & '0';

C <= '1' & '1' & '0';

A <= B & C; -- A now has "010110"

And Or gate in VHDL example



```
library IEEE;use IEEE.STD_LOGIC_1164.ALL;use IEEE.STD_LOGIC_ARITH.ALL;use IEEE.STD_LOGIC_UNSIGNED.ALL;entity ABorC is
```

```
    port (A : in std_logic; B : in std_logic;  
          C : in std_logic; F : out std_logic);
```

```
end ABorC;
```

```
architecture arch of ABorC is
```

```
    signal X : std_logic;
```

```
begin
```

```
    X <= A and B after 1 ns;
```

```
    F <= X or C after 1 ns;
```

```
end;
```

← Instead of the numeric package we will normally use the arithmetic package and the associated unsigned package. But the numeric package would work equally well.

Data Objects

A data object holds a value of a specified type.

Four classes of data objects are

- **Constant**

- hold a single value of a given type. This value cannot be changed during simulation.

- eg. **constant** rise_time:time:=10ns;

- **Variable**

- hold a single value of a given type. Different values can be assigned at different times.

- **local storage in process, procedures & functions**

- eg. **variable** sum:integer **range** 0 to 100:=10;

- variable** found:boolean;

Data Objects

- **Signal**

- holds a list of values which include the current value of the signal and a set of possible future values.
- **declared outside the process and can be used anywhere within the architecture.**

eg. **signal** clock:bit;

signal data_bus: bit_vector(0 to 7);

signal g_delay:time:=10ns;

Data Objects

- **File**

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

syntax:

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

eg: **file** stimulus:text open read_mode **is** “/user/home/add.sti”;

Data Flow Modeling

In this modeling style, the flow of data through the entity is expressed using concurrent (parallel) signal. The concurrent statements in VHDL are WHEN and GENERATE.

Besides them, assignments using only operators (AND, NOT, +, *, sll, etc.) can also be used to construct code.

Finally, a special kind of assignment, called BLOCK, can also be employed in this kind of code.

In concurrent code, the following can be used:

- ☐ Operators
- ☐ The WHEN statement (WHEN/ELSE or WITH/SELECT/WHEN);
- ☐ The GENERATE statement;
- ☐ The BLOCK statement

Dataflow Modeling

--program for half adder

Library ieee;

Use ieee.std_logic_1164.all;

Entity ha is

Port(a,b:in std_logic; sum,carry:out std_logic);

End ha;

Architecture halfadder of ha is

Begin

 sum<= a xor b;

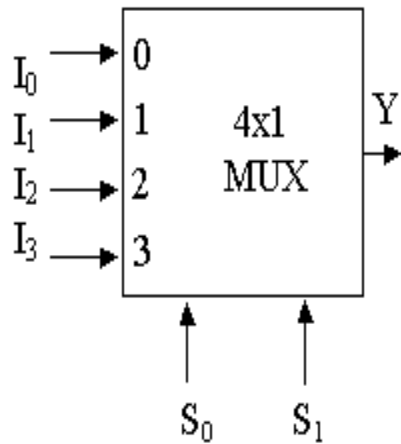
 carry<= a and b;

End halfadder;

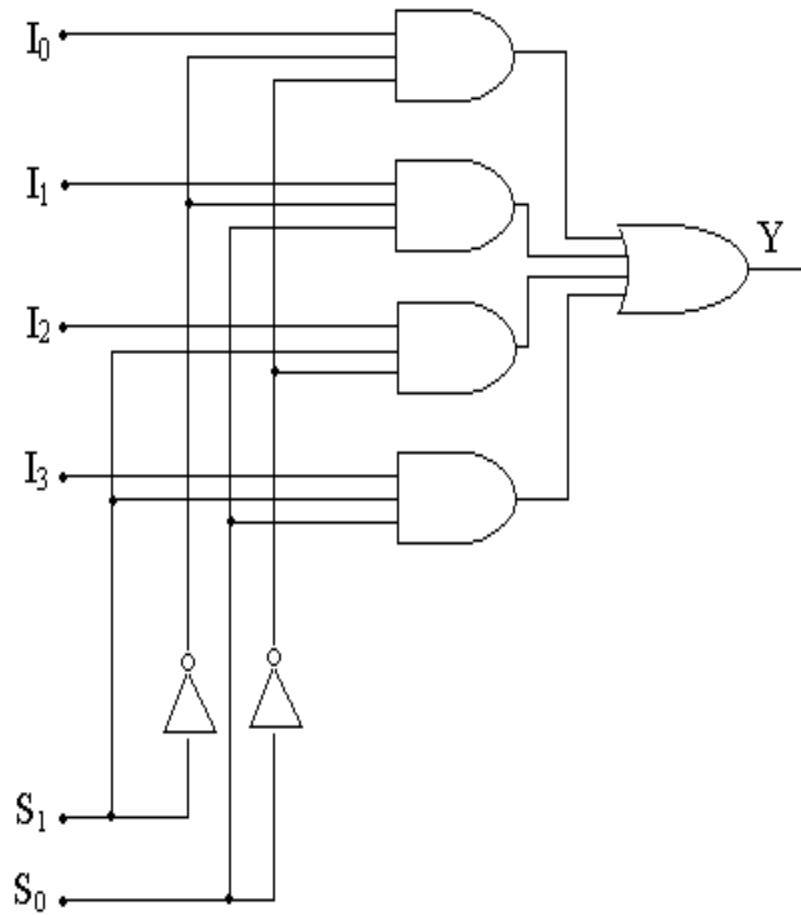
Concurrent Statements

- **Concurrent Signal Assignment Statement**
- **Conditional Signal Assignment Statement**
- **Selected Signal Assignment Statement**
- Block Statement
- Concurrent Assertion Statement

4:1 MUX



S_0	S_1	Y
0	0	0
0	1	1
1	0	2
1	1	3



Conditional Signal Assignment Statement

Syntax:

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

4:1 MUX using conditional signal assignment statement

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity mux_dat is
4     port(I:in std_logic_vector(0 to 3);
5           s:in std_logic_vector(1 downto 0);
6           y:out std_logic);
7 end mux_dat;
8 architecture muxdat of mux_dat is
9 begin
10     y<=I(0) when s="00" else
11         I(1) when s="01" else
12         I(2) when s="10" else
13         I(3);
14 end muxdat;
```

Selected Signal Assignment Statement

Syntax:

with expression select

```
target-signal<=waveform-elements when choices,  
      waveform-elements when choices,  
      .....  
      waveform-elements when choices;
```


4:1 MUX using selected signal assignment statement

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity mux_dat_sel is
4     port(I:in std_logic_vector(0 to 3);
5          s:in std_logic_vector(1 downto 0);
6          y:out std_logic);
7 end mux_dat_sel;
8 architecture muxdat of mux_dat_sel is
9 begin
10     with s select
11         y<=I(0) when "00",
12             I(1) when "01",
13             I(2) when "10",
14             I(3) when others;
15 end muxdat;
```

Concurrent Statements

Block statement:

- To disable signal drivers by using guards
- To limit scope of declarations
- To represent a portion of a design
- Any declaration appearing between **blockend block** are visible only within the block.

Block-label: **block** [(guard expression)] [**is**]

[block-header]

[block-declarations]

Begin

Concurrent – statements

End block[block-label];

Concurrent Assertion Statement

- Used to check if a signal value lies within a specified range.
- If check fails, a message is reported.
- Syntax:
assert boolean-expression
[**report** string-expression]
[**severity** expression]
- Predefined severity are NOTE, WARNING, ERROR and FAILURE.

Concurrent Assertion Statement

Example:

Architecture srflp of srff is

Begin

assert (S='1' and R='1')

report "Not valid inputs"

severity ERROR;

End srflp

Behavioral Modeling

In this modeling style, the behavior of an entity as set of statements is executed sequentially in the specified order.

Only statements placed inside a PROCESS, FUNCTION, or PROCEDURE are sequential.

PROCESSES, FUNCTIONS, and PROCEDURES are the only sections of code that are executed sequentially. However, as a whole, any of these blocks is still concurrent with any other statements placed outside it.

One important aspect of behavior code is that it is not limited to sequential logic. Indeed, with it, we can build sequential circuits as well as combinational circuits.

The behavior statements are IF, WAIT, CASE, and LOOP. VARIABLES are also restricted and they are supposed to be used in sequential code only. VARIABLE can never be global, so its value cannot be passed out directly.


Process

- The process is a fundamental concept in VHDL
- Processes are contained within an architecture.
 - An architecture may contain several processes.
- Processes execute concurrently (at the same time)
 - Code within a process execute sequentially.
- Combinational circuits can be modeled without using process.
- Sequential circuits need processes. Processes provide powerful statements not available outside processes.

Example

```
process (A, B)
  F <= A xor B;
end process;
```

Sensitivity list



Process statements occur within an architecture.

The syntax for a process is:

```
LABEL1:    -- optional label
process (signal_name, ...) is
-- declarations
begin
  -- sequential statements
end process LABEL1;
```

Behavioral Modeling

```
entity ha is
  port(a,b:in std_logic;sum,carry:out std_logic);
End ha;
Architecture half of ha is
Begin
  Process(a,b)
  begin
    if a='0' and b='0' then sum='0'; carry='0';
    elsif a='0' and b='1' then sum='1'; carry='0';
    elsif a='1' and b='0' then sum='1'; carry='0';
    elsif a='1' and b='1' then sum='0'; carry='1';
    else sum=X; carry=X;
    end if;
  End process;
End half;
```

```
--half ordder using Data
flow model
entity ha is
  port(a,b:in
std_logic;sum,carry:out
std_logic);
End ha;
Architecture half of ha is
Begin
  Sum <= a xor b;
  Carry <= a and b;
End ;
```

UP COUNTER DESIGN

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity counter is
    port(Clock, CLR : in std_logic;
          Q : out std_logic_vector(3 downto 0));
end counter;
architecture behav of counter is
    signal tmp: std_logic_vector(3 downto 0);
begin
    process (Clock, CLR)
    begin
        if (CLR='1') then
            tmp <= "0000";
        elsif (Clock'event and Clock='1') then
            tmp <= tmp + 1;
        end if;
    end process;
    Q <= tmp;
End behav;
```


Structural Modeling

In this modeling, an entity is described as a set of interconnected components. A component instantiation statement is a concurrent statement. Therefore, the order of these statements is not important. The structural style of modeling describes only an VLSI Design

interconnection of components (viewed as black boxes), without implying any behavior of the components themselves nor of the entity that they collectively represent.

In Structural modeling, architecture body is composed of two parts: the declarative part (before the keyword begin) and the statement part (after the keyword begin).

Structural Modeling-eg Half adder

```
Library ieee;  
Use ieee.std_logic_1164.all;  
Entity ha is  
Port(a,b:in std_logic; sum,carry:out std_logic);  
End ha;  
Architecture half_SM of ha is
```

Component xor2 –declarative part

port(p,q:in std_logic; r:out std_logic);

End component;

Component and2

port(x,y:in std_logic; z:out std_logic);

End component;

Begin

**--part_name: entity_name port map (actual
arguments) ; create instance syntax**

X1:xor2 port map(a,b,sum); --statement part

X2:and2 port map(a,b,carry);

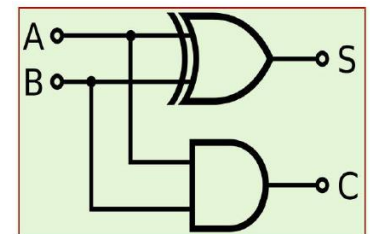
End half;

--xor two input gate

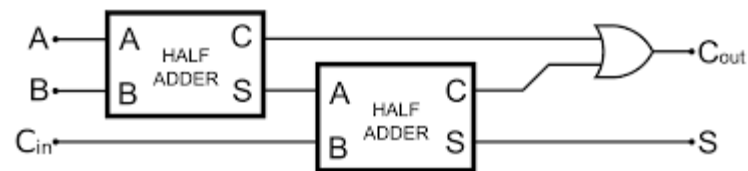
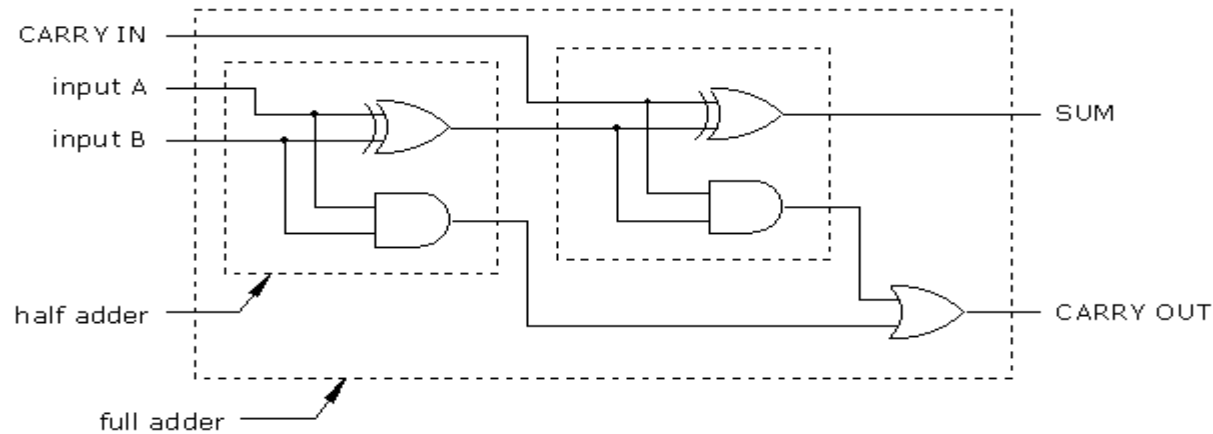
```
Library ieee;  
Use ieee.std_logic_1164.all;  
Entity xor2 is  
Port(p,q:in std_logic; r:out std_logic);  
End xor2;  
Architecture DF of xor2 is  
Begin  
r <= p xor q;  
End DF;
```

--and two input gate

```
Library ieee;  
Use ieee.std_logic_1164.all;  
Entity and2 is  
Port(x,y:in std_logic; z:out  
std_logic);  
End and2;  
Architecture DF of and2 is  
z <= x and y;  
End DF;
```



Full Adder using Half Adder



Full Adder using Half Adder

Library ieee;

Use ieee.std_logic_1164.all;

Entity fa is

port(x,y,z:in std_logic; s,c:out std_logic);

End fa;

Architecture fa_str of fa is

Component ha

port(a,b:in std_logic; sum,carry:out std_logic);

End component;

Component or2

port(p,q:in std_logic; r:out std_logic);

End component;

Signal s1,c1,c2:std_logic;

Begin

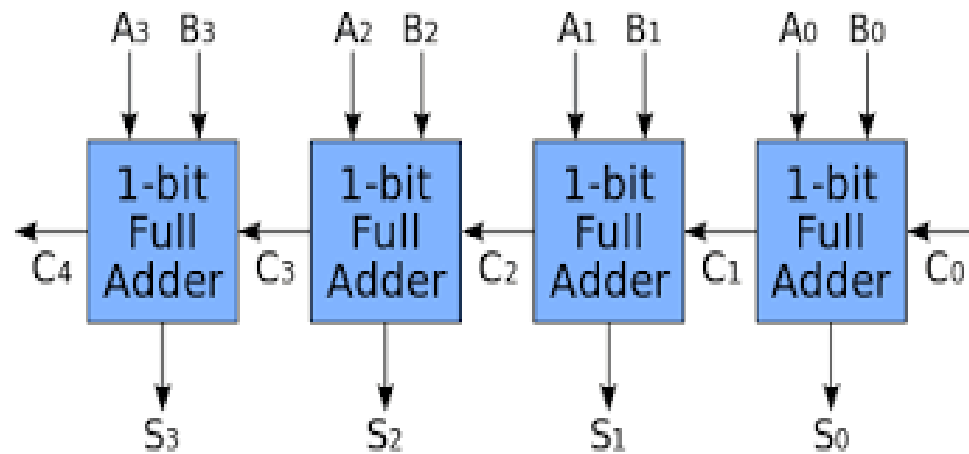
Fa1:ha port map(x,y,s1,c1);

Fa2:ha port map(s1,z,s,c2);

Fa3:or2 port map(c1,c2,c);

End fa_str;

4-bit Ripple Carry Adder using Full Adder



```

Library ieee;
Use ieee.std_logic_1164.all;

Entity ripple_str is
    port(A:in std_logic_vector(3 downto 0);
          B:in std_logic_vector(3 downto 0);
          Cin:in std_logic;
          sum:out std_logic_vector(3 downto 0);
          Cout:out std_logic);
End ripple_str;

Architecture ripple_str_fa of ripple_str is
    Signal Cint:std_logic_vector(1 to 3);

    Component fa
        port(x,y,z:in std_logic; s,c:out std_logic);
    End component;

```

Begin

```

rip1:fa port
    map(A(0),B(0),Cin,sum(0),Cint(1));
    rip2:fa port
    map(A(1),B(1),Cint(1),sum(1),Cint(2));
    rip1:fa port
    map(A(2),B(2),Cint(2),sum(2),Cint(2));
    rip1:fa port
    map(A(3),B(3),Cint(3),sum(3),Cout);
End ripple_str_fa;

```

(or)

```

rip1:fa port map(x=>A(0), y=>B(0), z=>Cin,
    s=>sum(0), c=>Cint(1);

```

Sequential Statements

- Variable-assignment statements
- Signal assignment
- Wait
- If
- Case
- Loop
- Null
- Exit, next, assertion, report, return, procedure-call

Case statement

- Syntax:

case expression **is**

when choices=>sequential-statements

when choices=>sequential-statements

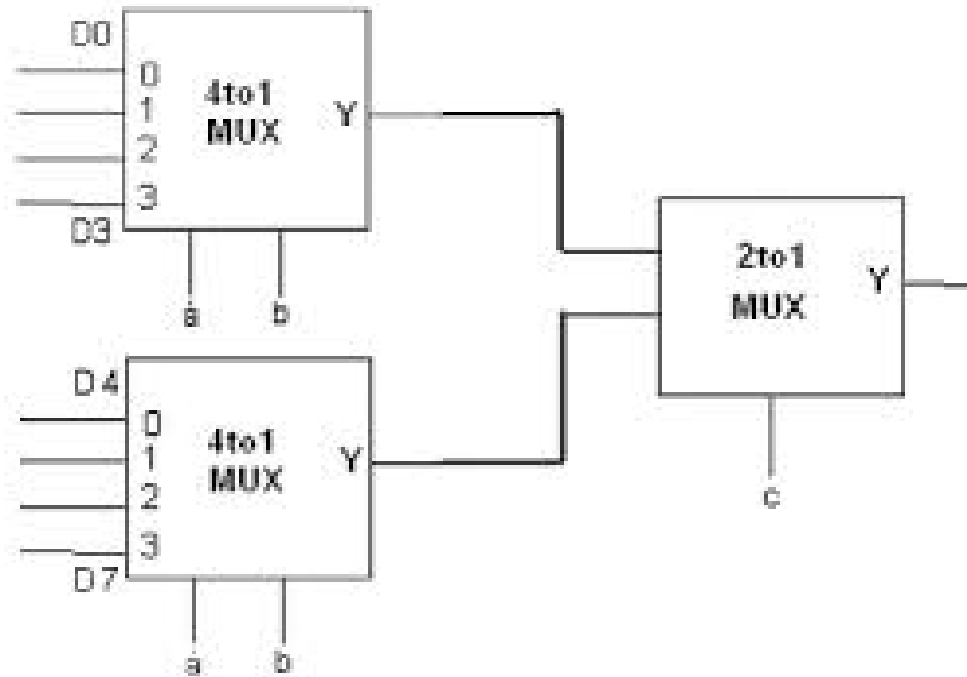
 [**when others**=>sequential-statements]

end case;

4:1 MUX using case

```
1 library ieee ;
2 use ieee.std_logic_1164.all ;
3 entity mux_case is
4     port(I:in std_logic_vector(0 to 3);
5           s:in std_logic_vector(1 downto 0);
6           y:out std_logic);
7 end mux_case;
8 architecture muxx of mux_case is
9 begin
10 process(I)
11 variable x:std_logic;
12 begin
13 case s is
14     when "00"=>x:=I(0);
15     when "01"=>x:=I(1);
16     when "10"=>x:=I(2);
17     when others=>x:=I(3);
18     --when others=> y<="----";
19 end case;
20 y<=x;
21 end process;
22 end muxx;
```

8:1 MUX using 4:1 MUX



8:1 MUX using 4:1 MUX

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity 8to1mux is
4      port(ip:in std_logic(0 to 7);
5            sel:in std_logic(2 downto 0);
6            yo:out std_logic);
7  end 8to1mux;
8  architecture 8to1 of 8to1mux is
9
10     component mux_case is
11         port(I:in std_logic_vector(0 to 3);
12               s:out std_logic_vector( 1 downto 0);
13               y:out std_logic);
14     end component;
15
16     component 2to1mux is
17         port(I:in std_logic_vector(0 to 1);
18               s:in std_logic;
19               y:out std_logic);
20     end component;
21     signal s1,s2:std_logic;
22     begin
23
24         m1:mux_case port map(ip(0),ip(1),ip(2),ip(3),sel(2),sel(1),s1);
25         m2:mux_case port map(ip(4),ip(5),ip(6),ip(7),sel(2),sel(1),s2);
26         m3:2to1mux port map(s1,s2,sel(0),yo);
27
28     end 8to1;
```

Sequential Statements - Wait

- an alternative to a sensitivity list
- a process cannot have both wait st. and a sensitivity list
- Generic form of a process with wait statement(s)

Process

begin

sequential-statements

wait statement

sequential-statements

wait-statement

...

end process;

- How wait statements work?

Execute seq. statement until wait statement is encountered.

Wait until the specified condition is satisfied.

When the end of the process is reached start over again at the beginning.

Sequential Statements - Wait

➤ **wait on** sensitivity-list;

- until one of the signals in the sensitivity list changes
- Eg. **Wait on A,B;**

Wait on clk **for** 20 ns;

wait for time-expression;

- waits until the time specified by the time expression has elapsed
- What is this:

wait for 10 ns;

➤ **wait until** boolean-expression;

- the boolean expression is evaluated whenever one of the signals in the expression changes, and the process continues execution when the expression evaluates to TRUE.
- **Wait until** sum>100 **for** 50 ns;
- **Wait until A=B;**

Sequential Statements - Loop

- The loop label is optional. By defining the range the direction as well as the possible values of the loop variable are fixed.
- The loop variable is only accessible within the loop.
- For synthesis the loop range has to be locally static and must not depend on signal or variable values.
- While loops are not generally synthesizable.
- Three kinds of iteration statements.

[label:] **loop**

sequence-of-statements -- use exit statement to get out

end loop [label] ;

[label:] **for** variable **in** range **loop**

sequence-of-statements

end loop [label] ;

[label:] **while** condition **loop**

sequence-of-statements

end loop [label] ;

Sequential Statements - Loop

- Egs.

1. fact:=1;

for no in 2 to N loop

fact:=fact*no;

end loop;

2. j:=0;sum:=10;

wh_lp:**while j<20 loop**

sum:=sum*2; j:=j+3;

end loop;

Sequential Statements - Next

A statement that may be used in a loop to cause the next iteration.

```
[ label: ] next [ label2 ] [ when condition ] ;
```

```
next;
```

```
next outer_loop;
```

```
next when A>B;
```

```
next this_loop when C=D or done;
```

-- done is a Boolean variable

Sequential Statements – Exit & Null

- A statement that may be used in a loop to immediately exit the loop.

[label:] exit [label2] [when condition] ;

exit;

exit outer_loop;

exit when A>B;

exit this_loop when C=D or done;

-- done is a Boolean variable

- Null
 - does not cause any action to take place; execution continues with the next statement.

Mixed Style of Modeling

```
1  entity fa is
2      port(a,b,c:in bit;s,c:out bit);
3  end fa;
4  architecture fa_mxd of fa is
5      component xor2
6          port(p,q:in bit;r:out bit);
7      end component;
8      signal s1:bit;
9  begin
10     --structural
11     x1:xor2 port map(a,b,s1);
12     --behavioral
13     process(a,b,c)
14         variable t1,t2,t3:bit;
15     begin
16         t1:=a and b;
17         t2:=b and c;
18         t3:=c and a;
19         c<=t1 or t2 or t3;
20     end process;
21     --data flow
22     s<=s1 xor c;
23 end fa_mxd;
```

Program for Sequential Circuits – D Flip-Flop

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity dff is
4  port(d,clk:in std_logic;
5        q:out std_logic);
6  end dff;
7  architecture dffl of dff is
8  begin
9      process(d,clk)
10     begin
11         if clk='1' then
12             q<=d;
13         else
14             q<='0';
15         end if;
16     end process;
17 end dffl;
18
```

D Flip-Flop

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity dff is
4  port(d,clk:in std_logic;
5        q,qb:out std_logic);
6  end dff;
7  architecture dff1 of dff is
8  begin
9      process(d,clk)
10         variable s,s1:std_logic;
11         begin
12             if (clk='1' and clk'event) then
13                 s:=d;
14                 s1:= not(s);
15             else
16                 s:='0';
17                 s1:=not(s);
18             end if;
19             q<=s;qb<=s1;
20         end process;
21     end dff1;
22
```

JK Flip-Flop-Using if-else

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity jkff is
4      port(j,k,clk:in std_logic;
5            q,qb:out std_logic);
6  end jkff;
7  architecture jkff1 of jkff is
8      begin
9          process(j,k,clk)
10             variable s,s1:std_logic;
11             begin
12                 if (clk='1' and clk'event) then
13                     if (j='0' and k='0') then
14                         s:=s;s1:=not(s);
15                     elsif (j='0' and k='1') then
16                         s:='0';s1:=not(s);
17                     elsif (j='1' and k='0') then
18                         s:='1';s1:=not(s);
19                     elsif (j='1' and k='1') then
20                         s:=not(s);s1:=not(s);
21                     end if;
22                 else
23                     s:=s;s1:=not(s);
24                 end if;
25                 q<=s;qb<=s1;
26             end process;
27  end jkff1;
```

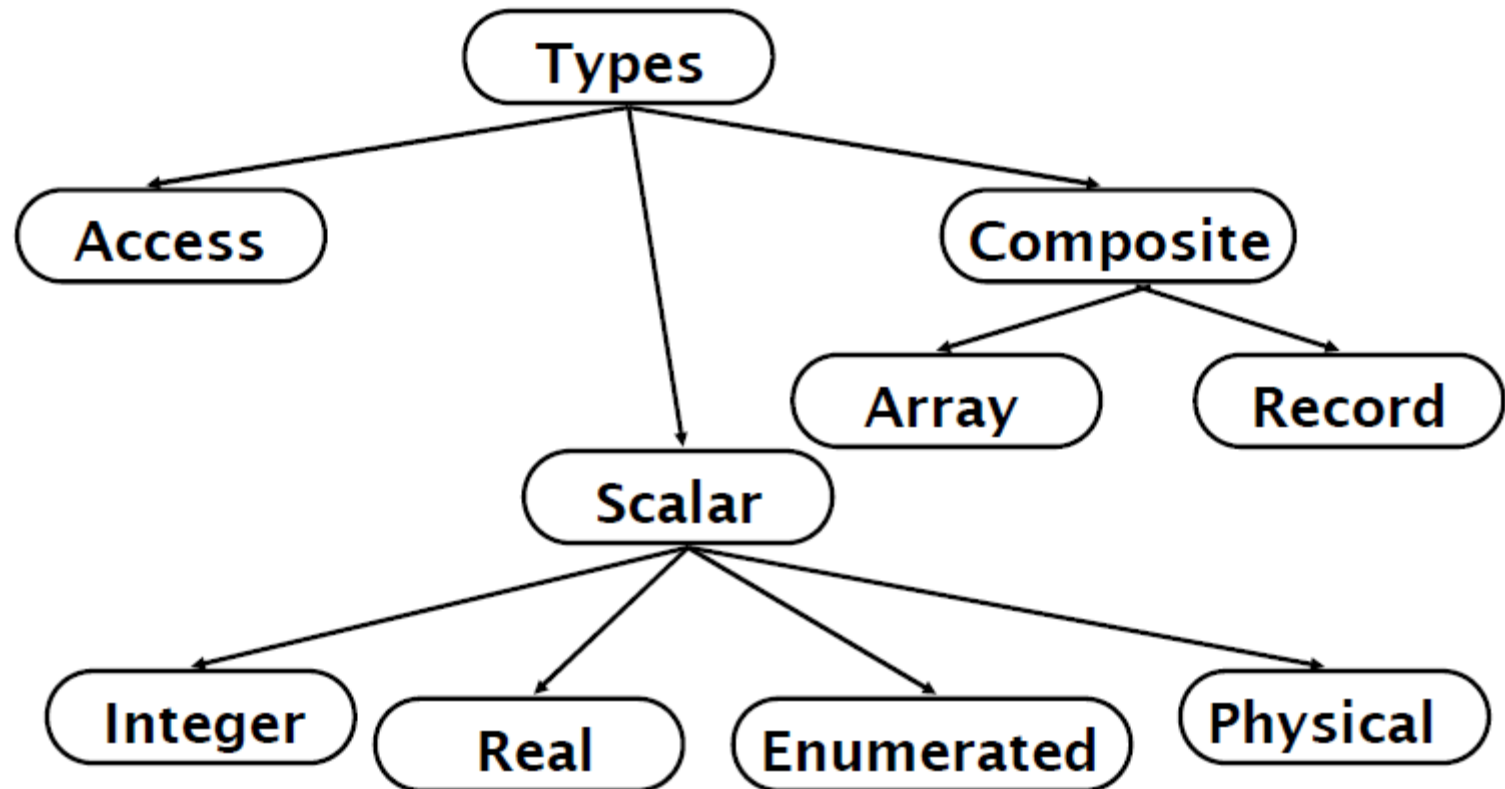
JK Flip-Flop-using case statement

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity jkff1 is
4  port(j,k,clk:in std_logic;
5       q,qb:out std_logic);
6  end jkff1;
7  architecture jkk of jkff1 is
8  signal s:std_logic_vector(0 to 1);
9  begin
10  s<=j&k;
11  process(clk)
12  variable s1,s2:std_logic;
13  begin
14  if (clk='1' and clk'event) then
15  case s is
16  when "00"=>s1:=s1; s2:=not(s1);
17  when "01"=>s1:='0'; s2:=not(s1);
18  when "10"=>s1:='1'; s2:=not(s1);
19  when others=>s1:=not(s1); s2:=not(s1);
20  end case;
21  end if;
22  q<=s1;qb<=s2;
23  end process;
24  end jkk;
```

4-bit Up Counter

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  entity upcount is
5      port (clk,R:in std_logic;
6            Q:out std_logic_vector(0 to 3));
7  end upcount;
8  architecture cou of upcount is
9      -- signal count:std_logic_vector(0 to 3);
10     begin
11         process (R,clk)
12             variable count:std_logic_vector(0 to 3);
13             begin
14                 if R='1' then
15                     count:="0000";
16                 elsif (clk='1' and clk'event) then
17                     count:=count+1;
18                 end if;
19                 q<=count;|
20             end process;
21             --q<=count;
22     end cou;
```

VHDL Data Types



Predefined Data Types

- bit ('0' or '1')
- bit_vector (array of bits)
- integer
- real
- time (physical data type)

Integer

- Integer
 - Minimum range for any implementation as defined by standard:
-2,147,483,647 to 2,147,483,647
 - Integer assignment example

```
ARCHITECTURE test_int OF test IS
BEGIN
    PROCESS (X)
        VARIABLE a: INTEGER;
    BEGIN
        a := 1;  -- OK
        a := -1; -- OK
        a := 1.0; -- bad
    END PROCESS;
END TEST;
```

Real

- Real
 - Minimum range for any implementation as defined by standard: -1.0E38 to 1.0E38
 - Real assignment example

```
ARCHITECTURE test_real OF test IS
BEGIN
  PROCESS (X)
    VARIABLE a: REAL;
  BEGIN
    a := 1.3;  -- OK
    a := -7.5; -- OK
    a := 1;    -- bad
    a := 1.7E13; --OK
    a := 5.3 ns; -- bad
  END PROCESS;
END TEST;
```

Enumerated

- Enumerated
 - User defined range
 - Enumerated example

```
TYPE binary IS ( ON, OFF );  
... some statements ...  
ARCHITECTURE test_enum OF test IS  
BEGIN  
    PROCESS (X)  
        VARIABLE a: binary;  
    BEGIN  
        a := ON;  -- OK  
        ... more statements ...  
        a := off; -- OK  
        ... more statements ...  
    END PROCESS;  
END TEST;
```

```
-----  
-- logic state system (unresolved)  
-----  
TYPE std_ulogic IS ( 'U', -- Uninitialized  
                     'X', -- Forcing Unknown  
                     '0', -- Forcing 0  
                     '1', -- Forcing 1  
                     'Z', -- High Impedance  
                     'W', -- Weak Unknown  
                     'L', -- Weak 0  
                     'H', -- Weak 1  
                     '-' -- Don't care  
);
```

Physical

- Physical
 - Can be user defined range
 - Physical type example

```
TYPE resistance IS RANGE 0 to 1000000

UNITS
    ohm;    -- ohm
    Kohm = 1000 ohm;    -- 1 KΩ
    Mohm = 1000 kohm;    -- 1 MΩ
END UNITS;
```

- Time units are the only predefined physical type in VHDL.

Array

- Array
 - Used to collect one or more elements of a similar type in a single construct.
 - Elements can be any VHDL data type.

```
TYPE data_bus IS ARRAY (0 TO 31) OF BIT;
```

0...element numbers... 31

0	...array values...	1
----------	---------------------------	----------

```
VARIABLE X:  data_bus;  
VARIABLE Y:  BIT  
  
Y := X(12);  -- Y gets value of 12th element
```

```
TYPE register IS ARRAY (15 DOWNT0 0) OF BIT;
```

Record

- Record
 - Used to collect one or more elements of different types in a single construct.
 - Elements can be any VHDL data type.
 - Elements are accessed through field name.

```
TYPE binary IS ( ON, OFF );
TYPE switch_info IS
    RECORD
        status : binary;
        IDnumber : integer;
    END RECORD;

VARIABLE switch : switch_info;

switch.status := on;  -- status of the switch
switch.IDnumber := 30; -- number of the switch
```

Subtype

- Subtype
 - Allows for user defined constraints on a data type.
 - May include entire range of base type.
 - Assignments that are out of the subtype range result in error.
 - Subtype example

```
SUBTYPE name IS base_type RANGE <user range>;
```

```
SUBTYPE first_ten IS INTEGER RANGE 0 to 9;
```


Natural and Positive Integers

- Integer subtypes:
 - Subtype Natural is integer range 0 to integer'high;
 - Subtype Positive is integer range 1 to integer'high;

Boolean, Bit and Bit_vector

- type Boolean is (false, true);
- type Bit is ('0', '1');
- type Bit_vector is array (integer range <>) of bit;

Char and String

- type Char is (NUL, SOH, ..., DEL);
 - 128 chars in VHDL'87
 - 256 chars in VHDL'93
- type String is array (positive range <>) of Char;

ALGORITHMIC STATE MACHINE

- It is a sequential network which controls a digital system that carries out a step-by-step procedure or algorithm.

Components

1. State Box :

- Contains output list.

2. Decision Box : Condition is placed.

3. Conditional Output Box : Contains conditional output list.

4. ASM Block : contains one or more exit paths. Describes machine operation

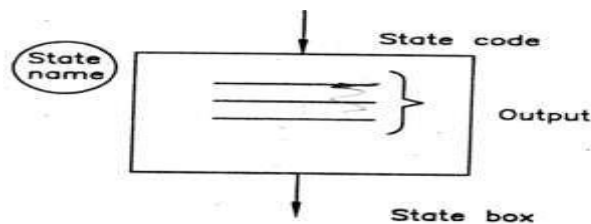


Fig. 4.1. State Box.

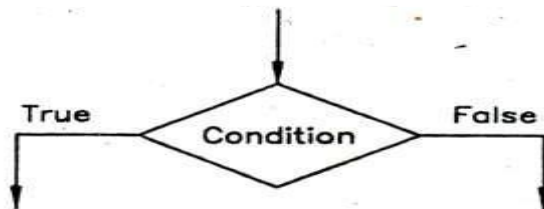


Fig. 4.2. Decision Box.

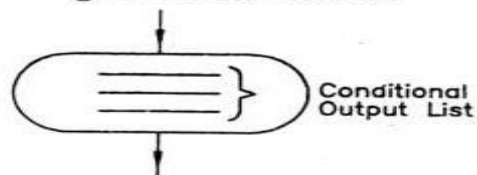


Fig. 4.3. Conditional Output box.

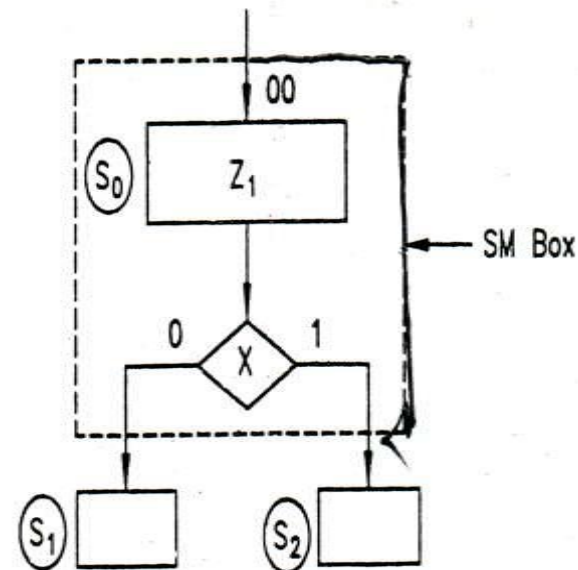


Fig. 4.4. ASM Block.

ASM chart for combinational circuits

Contains only one state box For $Z = A + BC$ Put $(A + BC)$ in one decision box & Z in one Conditional output box

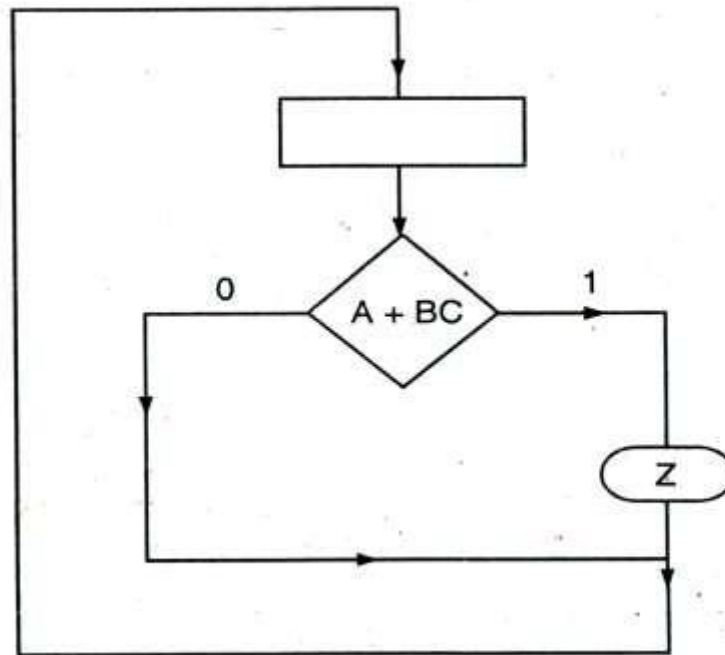


Fig. 4.5.

- Put each term in condition box If $A=1, Z=1$
If $A=0$ ($B=0, Z=0$ or $B=1, C=0/1, z=0/1$)

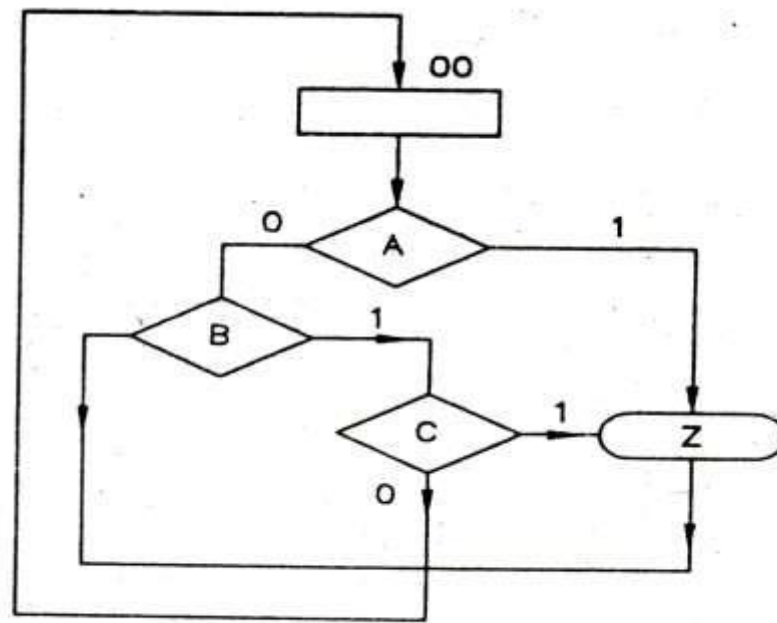


Fig. 4.6.

Sequential machine

There are two types of machines

1. MEALY MACHINE 2. MOORE MACHINE

1. MAELY MACHINE: Output is function of both input & present state

• Steps for making ASM chart

- i) Represent state by state boxes.
- ii) After each box put input in each box.
- iii) Depending on values of output the output in conditional box in paths where it will be "1".
- iv) iv) Depending on values of input connect the path to next state box.

Example of development of ASM chart from Mealy Machine Consider the Mealy state graph of 111 detector.

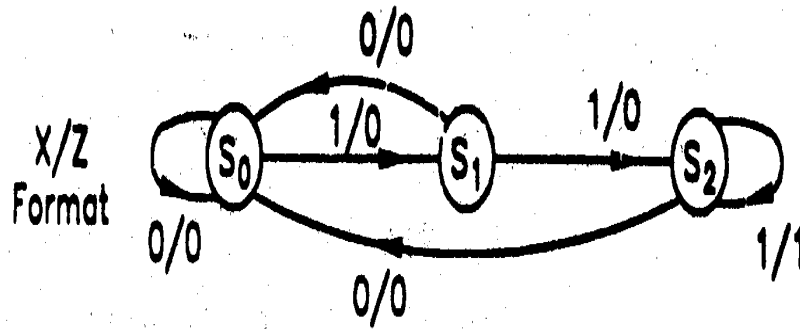


Fig. 4.8.

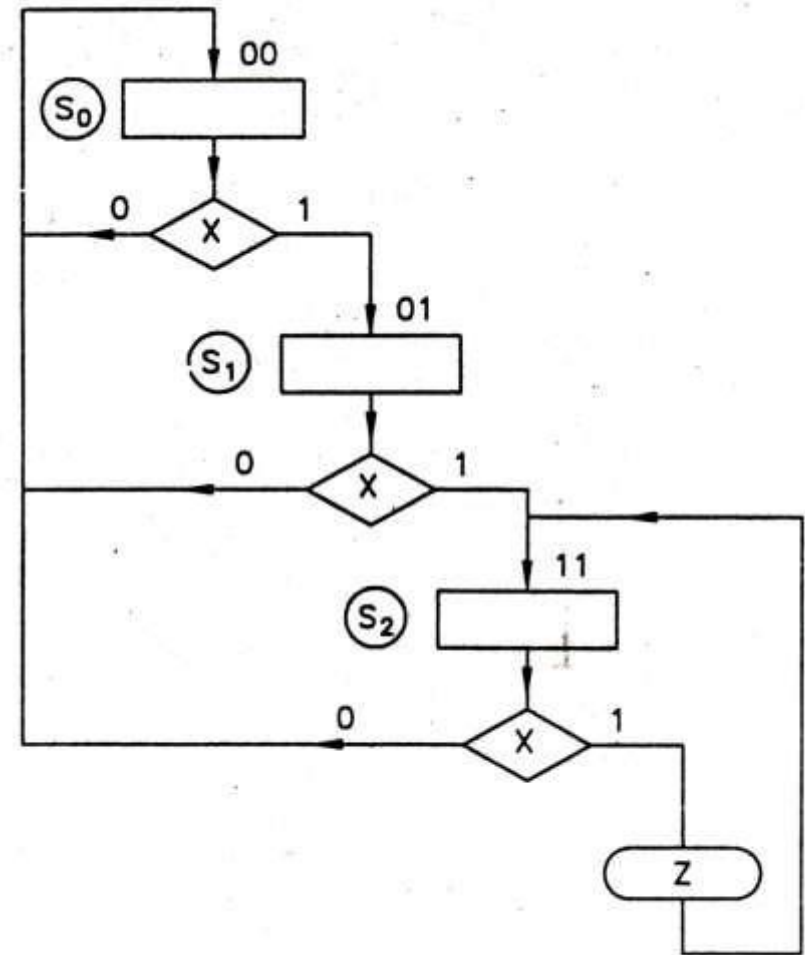


Fig. 4.12. SM chart, for 111 sequence detector.

2. MOORE MACHINE : Output depends on present state

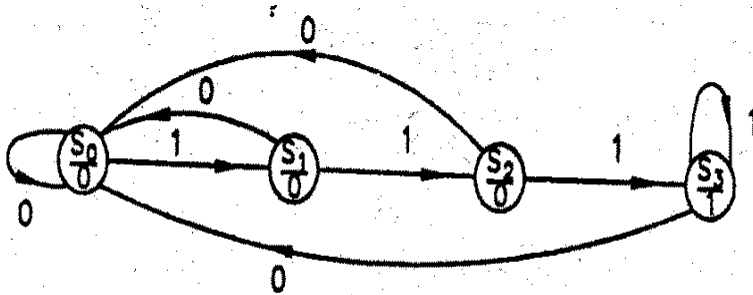


Fig. 4.13.

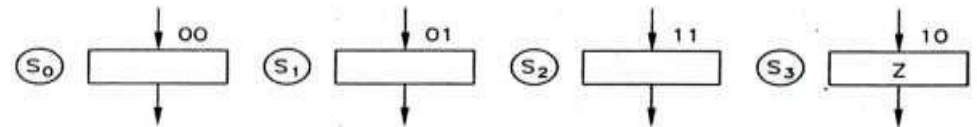


Fig. 4.14.

Now put the input in decision box and connect next state links.

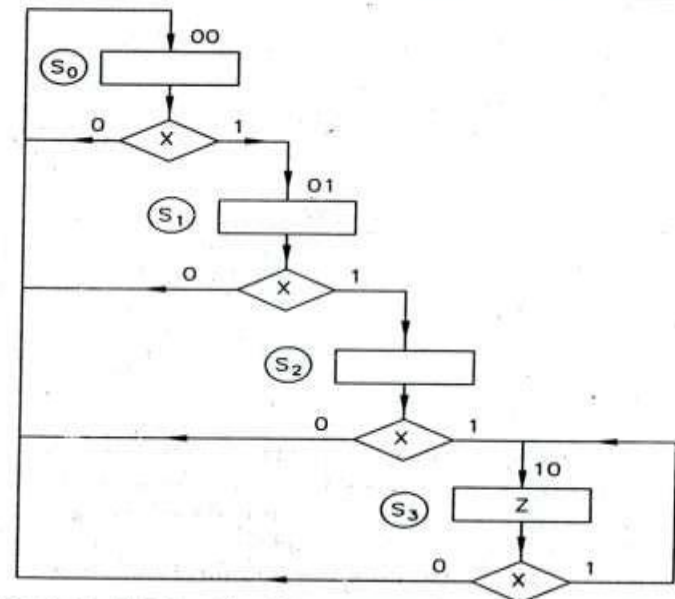


Fig. 4.15. SM chart for 111 sequence detector Moore circuit.

ASM -Advantage

- Easy to understand.
- May have one or more equivalent forms.
- Can describe both combinational & sequential circuits.
- Have structured approach to visualise a sequential problem.