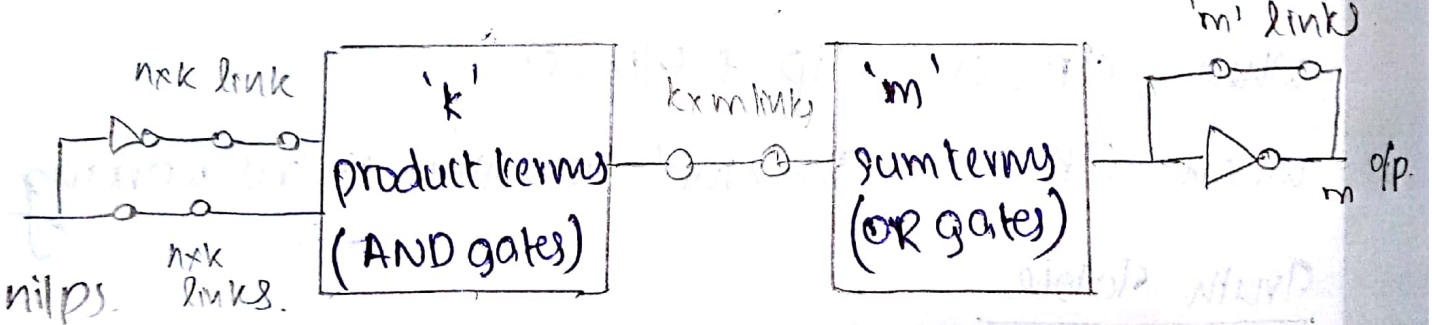
14/11/19

MODULE - 6

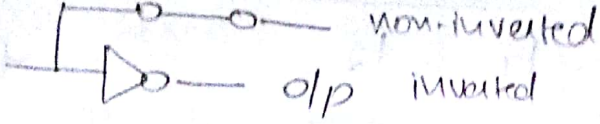
Programmable Logic Array (PLA)

- PLA is used to overcome drawbacks of ROM.
- If we have 100 o/p lines & we need only 2 others are waste. ie, ROM is not considering don't care.
- In PLA we are designing according to what we need.
- In PLA no decoders.
- Decoders in ROM is replaced by AND.

Block Diagram of PLA



n no:of i/p
k no:of AND gates.
To each AND gate we are giving inverted & non inverted i/p.
The o/p of these AND gates is feed as i/p to OR gates.
m OR gates.
∴ m o/p lines.

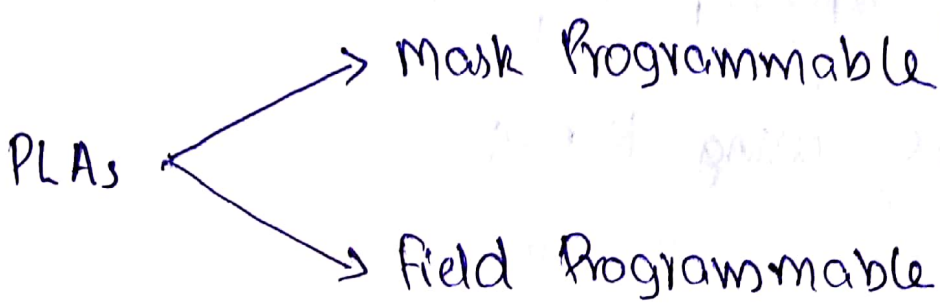


This is logically correct.

The size of PLA is determined by no: of i/p, no: of product terms, no: of o/p/OR (n x k x m)

? 2 x 3 x 5

2 i/ps ; 3 AND gates ; 5 OR gates.



→ Mask Programmable

We will give our functions in a table.

It is called PLA Program table.

We will give this PLA program table & they will give PLA. by making necessary connections

→ Field Programmable

By using certain devices, techniques & methods we will make PLAs

For a problem we will be given truth table

A	B	C	F_1	f_2
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	1	1
1	1	0	0	0
1	1	1	1	1

$F(A,B,C)$

$F = \Sigma(0, \ldots)$

Or

Minimize using K map.

A \ BC	00	01	11	10
0	0	0	0	0
1	1	1	1	0

$F_1 = AB' + AC$

A \ BC	00	01	11	10
0	0	0	1	0
1	0	1	1	0

$F_2 = AC + BC$

3 : I/Ps 2 : O/Ps.

$\therefore$ 3 x k x 2

- If we have simplified expression find out distinct product terms
- The absence of particular I/P can be put as —

PLA table consist of 3 columns.

1st column : It lists the product terms numerically

2nd column : It specifies the required paths b/w i/ps & AND gates

3rd column : It specifies the paths b/w the AND gates & OR gates.

PLA Program Table

	Product Terms.	Inputs			Outputs	
		A	B	C	F_1	F_2
AB'	1	1	0	–	1	–
AC	2	1	–	1	1	1
BC	3	–	1	1	–	1
					T	T

since in F_1 eqn there is no BC

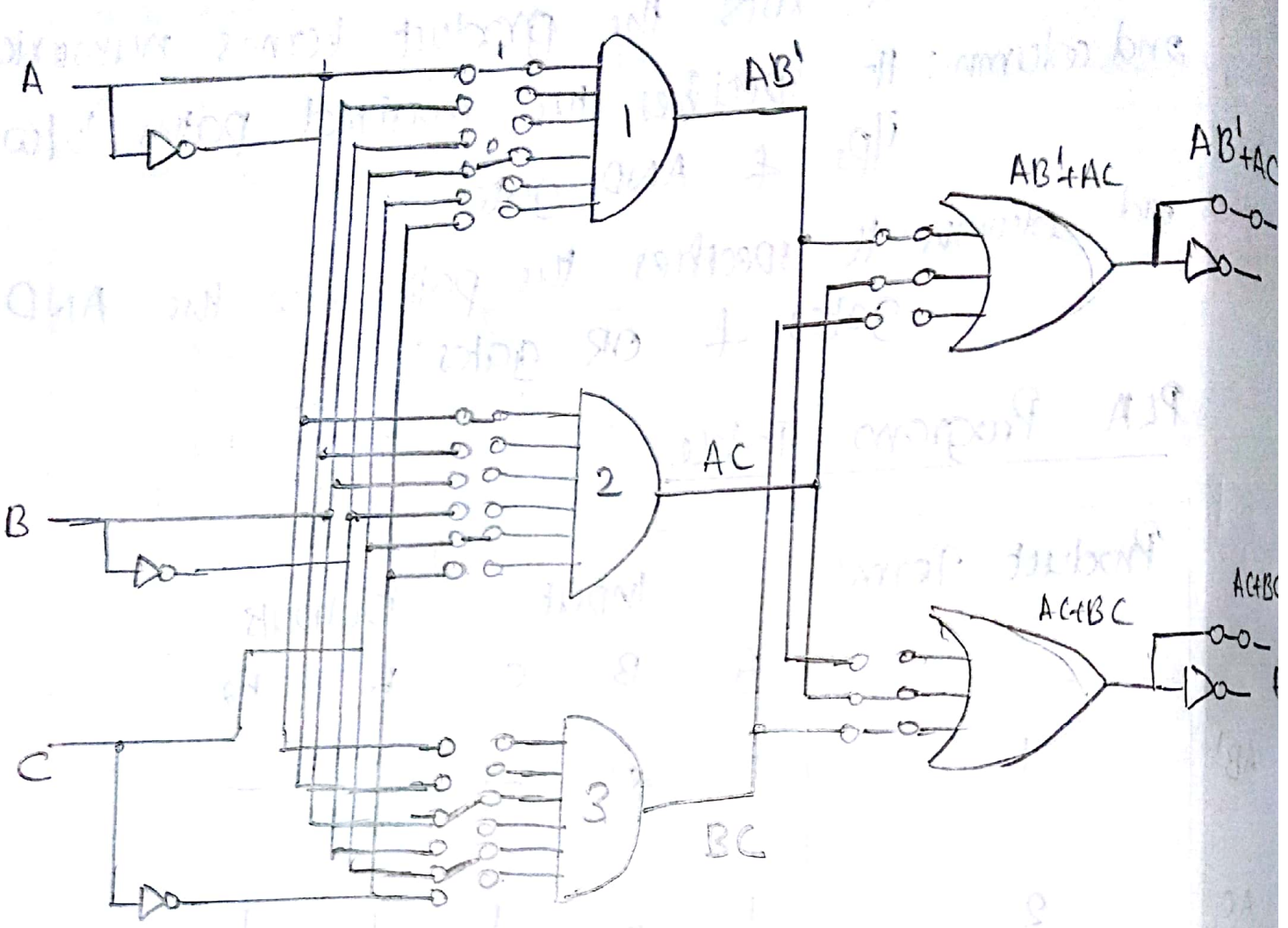
In truth table look where F_1 is one & fill A B C

If there is no inversion put T

To fill o/ps F_1 & F_2.

To fill F_1 look 1 in truth table, if there AB' is coming put 1 else put –

for another types of questions, simply using POS. & SOP. and take.



16/11/17.

HDL Programming.

- Hardware Description language.

Two types of languages in HDL

- Verilog
- VHDL

VHDL : more prominant.

Verilog.

- Verilog is in terms of modules.
- Operations → NOT, & AND, | OR
- signal : i/p, o/p, interconnection blw
  It can be 0's & 1's.

VHDL

↳ VHSIC Hardware Description Language

↳ Very High Speed Integrated Circuit

→ Program structure of VHDL.

1) Library
2) Entity
3) Architecture

Library.

```
Library IEEE;
Use ieee.std_logic_1164.all;
```

In VHDL each statement terminate with ;

use this package.

Use ieee.std_logic_1164.all;

↓ use IEEE lib

use all packages

Entity

- building blocks.
- declares i/p & o/p.

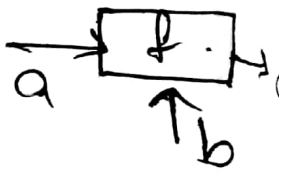
entity abcd is
↓ keyword ↓ name

> VHDL is case insensitive

entity abcd is

port (a: in std_logic;
b: in std_logic;
c: out std_logic
);

end abcd;



Let f = AND

Architecture

> Code of entity
> writes the code of the progm.

architecture arc_abcd of abcd is
(→ same name of entity)

begin

c <= a and b;
(→ assigning (=) operation)

end arc_abcd;

Modelling the VHDL Programms.

VHDL Programs can be done with 3 different styles.

Modelling Styles in VHDL

1) Data flow modelling
2) Behavioral modelling
3) Structural modelling

Data flow Modelling

- It shows that how the data or signal flows from i/p to o/p. through the registers or components.
- This works on concurrent execution

Behavioural Modelling

- Shows how our system performs according to current i/p values
- Here we define that what value we get at the o/p corresponding to i/p values.
- Defines the function or behaviour of our digital system.
- Works on sequential execution

Structural Modelling

- Shows the graphical representation of

modules or instances or components with their interconnection

- Defines that how our components or registers or modules are connected to each other
- Works on concurrent execution.

Data Flow Modelling - Full Adder

```
library IEEE;
use IEEE.std_logic_1164.all;                    } library
entity fab is
    port (a: in std_logic;
          b: in std_logic;
          c: in std_logic;
        sum: out std_logic;
      carry: out std_logic
      );
end fab;                                        } entity

architecture arc_fab of fab is
    begin
        sum <= a xor b xor c;
        carry <= (a and b) or
                 (b and c) or
                 (a and c);
end arc_fab;
```

Behavior Modelling - full adder.

process : It is the basic building block to work on behavior modelling style.

Truth table of full adder.

A	B	Cin	S	Cout.
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

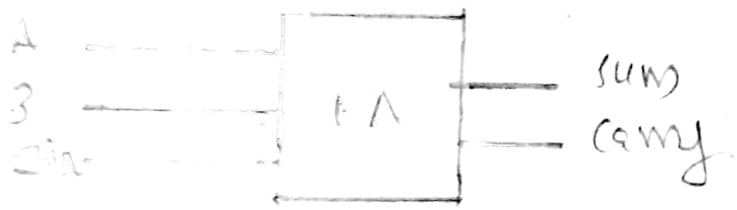
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
entity fab is
    port( a: in std_logic;
          b: in std_logic;
          c: in std_logic;
          s: out std_logic;
          cr: out std_logic
        );
end fab;
```

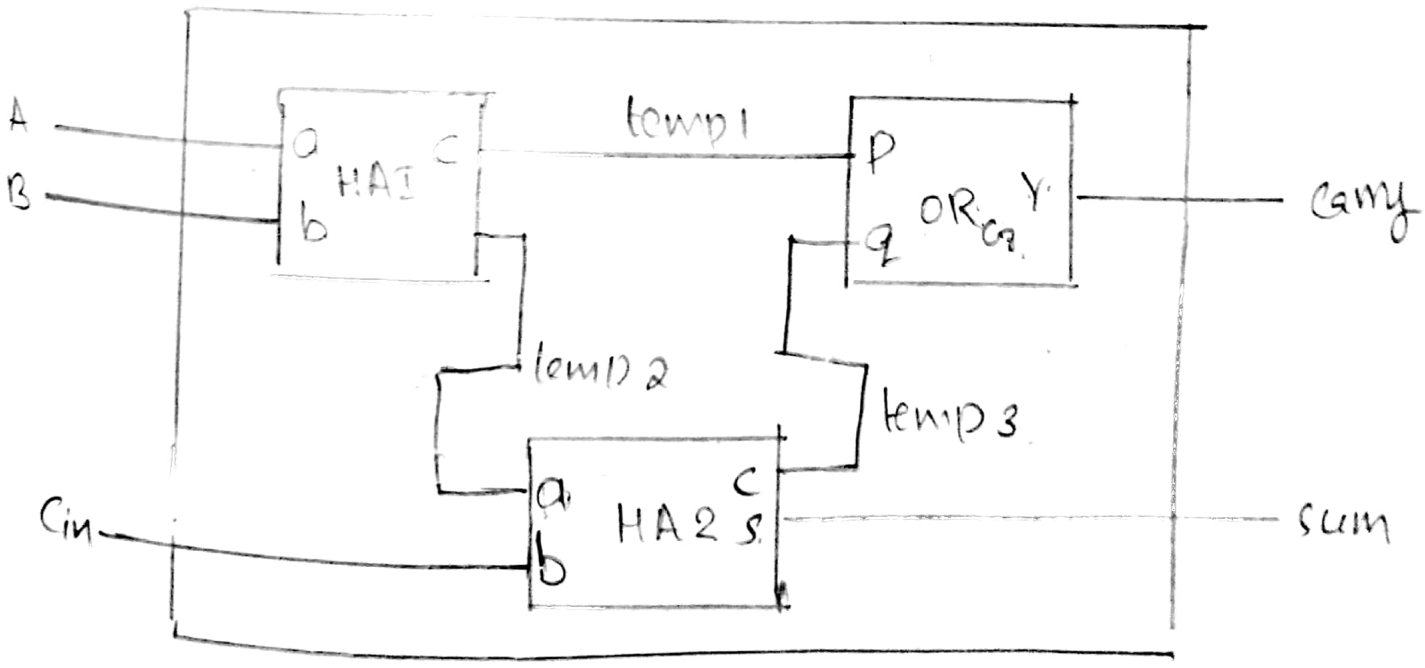
```
architecture arc_fab of fab is
begin
    process (a, b, c)
    begin
    if (a = '0' and b = '0' and c = '0') then
        s <= '0';
        cr <= '0';
    elsif (a = '0' and b = '0' and c = '1') then
        s <= '1';
        cr <= '0';
    elsif (a = '0' and b = '1' and c = '0') then
        s <= '1';
        cr <= '0';
    elsif (a = '0' and b = '1' and c = '1') then
        s <= 0;
        cr <= 1;
    elsif (a = '1' and b = '0' and c = '0') then
        s <= 1;
        cr <= 0;
    elsif (a = '1' and b = '0' and c = '1') then
        s <= 0;
        cr <= 1;
```

```
elsif (a = '1' and b = '1' and c = '0') then
        s <= 0';
        cr <= 1';

else
        s <= '1';
        cr <= '1';

end if;
end process;
end arc_fab;
```





temp1, temp2, temp3 → connection lines
We cannot connect components directly
we use connecting wires.

→ Write entity of system
→ In architecture, declare the signals & write components before 'begin' keyword.
→ After 'begin', do port mapping
→ End architecture
→ write entity for component
→ write architecture for component

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
  entity fadd is
     port( A: IN  STD_LOGIC;
           B: IN  STD_LOGIC;
          Cin: IN  STD_LOGIC;
          Sum: OUT  STD_LOGIC;
         Carry: OUT  STD_LOGIC
         );
  end fadd;
```

comment
-- architecture of your entity.

architecture arch_fa of fadd is.

-- declare signals.

Signal temp1, temp2, temp3 : std_logic;

-- write components.

component Hadd is

port (a,b : IN STD_LOGIC;
c,s : OUT STD_LOGIC;
);

end component;

component OR1 is

port (P, Q : in std_logic;
r : out std_logic
);

end component;

HA1 component made from Hadd

begin

HA1 : Hadd port map (a => A, b => B,
c => temp1, s => temp2);

HA2 : Hadd port map (a => temp2, b => cin
c => temp3, s => sum);

OR1 : OR1 port map (p => temp1 ; q => temp3,
r => carry);

end arch A.

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

entity Hadd is
  port( a,b: IN STD_LOGIC;
        c,s: IN STD_LOGIC
      );
end Hadd;

architecture arch_HA of HAdd is
begin
    s <= a xor b;
    c <= a and b;
end arch_HA;

library iee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all
entity OR1 is
    port( p,q: in std_logic;
          r : out std_logic;
        );
end OR1;
```

architechive arch_OR of OR1 is
begin

18/11/17 Inverter

Verilog

```
module inv (input [3:0] a,
            output [3:0] y);

  assign y = ~a;

endmodule
```

a is i/p having 4 bits

VHDL

```
library IEEE : use IEEE.STD_LOGIC_1164.all;
entity inv is
  port(a: in STD_LOGIC_VECTOR (3 downto 0);
       y: out STD_LOGIC_VECTOR (3 down to 0));

end;
architecture synth of ~~low~~ inv is
begin
    y <= not a;
end;
```

keyword

Data flow model is used here
This can also be done in behavior model.

Logic Gates

Verilog

```
module gates (input [3:0] a, b,
              output [3:0] y1, y2, y3, y4, y5);

/* Five different two-input logic gates acting
   on 4 bit buses */

assign y1 = a & b;   // AND
assign y2 = a | b;   // OR
assign y3 = a ^ b;   // XOR
assign y4 = ~(a & b); // NAND
assign y5 = ~(a | b); // NOR

endmodule
```

VHDL

```
library IEEE : use IEEE.STD_LOGIC_1164.all;
entity gates is.
  port(a, b: in STD_LOGIC_VECTOR(3 downto 0);
       y1, y2, y3, y4, y5: out STD_LOGIC_VECTOR
                              (3 downto 0));
end;
architecture synth of gates is.
begin.
-- five different two-input logic gates.
-- acting on 4 bit buses
```

```
y1 <= a and b;
y2 <= a or b;
y3 <= a xor b;
y4 <= a nand b;
yr <= a nor b;
end;
```

Multiplexer

Verilog

A 4:1 multiplexer can select one of 4 i/p using nested conditional operators.

```
module mux4 (input [3:0] d0,d1,d2,d3,
             input [1:0] s,
             output [3:0] y);
  assign y = s[1] ? (s[0] ? d3 : d2) : (s[0] ? d1:d0)
endmodule.
```

VHDL

A 4:1 multiplexer can select one of 4 i/p using multiple else clauses in the conditional signal assignment.

```
library IEEE : use IEEE.STD_LOGIC_1164.all;
entity mux4 is
  port (d0, d1,
        d2, d3 : in STD_LOGIC_VECTOR(3 downto 0);
        s : in STD_LOGIC_VECTOR (1 downto 0);
        y : out STD_LOGIC_VECTOR (3 downto 0));
 end;
architecture synth of mux4 is
begin
    y <= d0 when s = "00" else
         d1 when s = "01" else
         d2 when s = "10" else
         d3;
end;
```

Adder

Verilog

In Verilog, wires are used to represent internal variables when values are defined by assign statements such as assign p = a ^ b; Wires technically have to be declared only for multibit busses, but it is good practice to include them for all internal variables: their declaration could have been omitted in this example.

```
module fulladder (input a,b,cin,
                  output s,cout);

wire p,g;
assign p = a^b;
assign g = a&b;

assign s = p^cin;
assign cout = g | (p&cin);
endmodule
```

VHDL

In VHDL, signals are used to represen

Binary Addition

A B	SUM	CARRY
0 + 0	0	0
0 + 1	1	0
1 + 0	1	0
1 + 1	0	1

Binary Addition Algorithm.

Step 1: Read two n bit numbers

Step 2: Take LSB bit of first number & LSB of 2nd number & find sum & carry using the rule.

a) 0+0 = 0 — sum = 0, carry = 0

b) 0+1 = 1 — sum = 1, carry = 0

c) 1+0 = 1 — sum = 1, carry = 0

d) 1+1 = 10 — sum = 0, carry = 1

Step 3: If a carry is generated, it is to be propogated to next groups of bit and use the following rules for addition and repeat steps 2 & 3 for all bits.

1+0+0 = 01 — sum = 1, carry = 0

1+0+1 = 10 — sum = 0, carry = 1

1+1+0 = 10 — sum = 0, carry = 1

1+1+1 = 11 — sum = 1, carry = 1

Binary subtraction

A	B	Diff	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Binary subtraction - Algorithm

Step 1: Read two n bit numbers

Step 2: Take LSB bit of first number & LSB of 2nd number & check for the following conditions

0 - 0 = 0

1 - 1 = 0

1 - 0 = 1

10 - 1 = 1 0 - 1 with a borrow of 1

~~0 - 1~~

Step 3: If a borrow is needed, take 1 from next column to left & repeat steps 2 & 3 for all bits.