Part-1

Assignment 2

This chapter explains the VHDL programming for Combinational Circuits.

Objective:

(i) To construct half and full adder circuit and verify its working
(ii) To construct half and full subtractor circuit and verify its working
(iii) To construct a full adder-subtractor circuit

Half adder:

Let's start with a half (single-bit) adder where you need to add single bits together and get the answer. The way you would start designing a circuit for that is to first look at all of the logical combinations. You might do that by looking at the following four sums:

0 0 1 1

+ 0 + 1 + 0 + 1

0 1 1 10

That looks fine until you get to 1 + 1. In that case, you have a carry bit to worry about. If you don't care about carrying (because this is, after all, a 1-bit addition problem), then you can see that you can solve this problem with an XOR gate. But if you do care, then you might rewrite your equations to always include 2 bits of output, like this:

0 0 1 1

+ 0 + 1 + 0 + 1

00 01 01 10

Now you can form the logic table:

1-bit Adder with Carry-Out

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

By looking at this table you can see that you can implement the sum Q with an XOR gate and C (carry-out) with an AND gate.

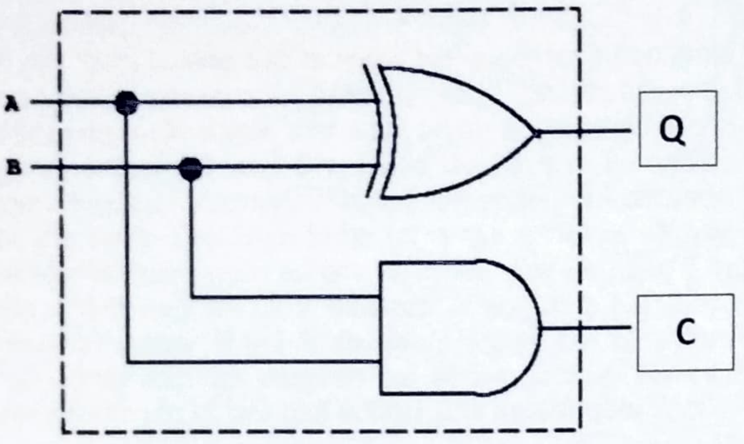
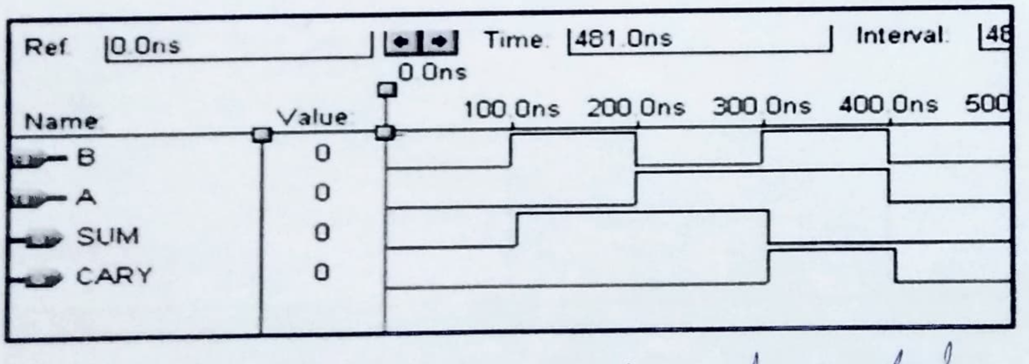


Fig. 1: Schematics for half adder circuit

VHDL Code for a Half-Adder

```
VHDL Code:

Library ieee;
use ieee.std_logic_1164.all;

entity half_adder is
   port(a,b:in bit; sum,carry:out bit);
end half_adder;

architecture data of half_adder is
begin
   sum<= a xor b;
   carry <= a and b;
end data;
```

Waveforms



Also write code for behavioural model.

Full adder:

If you want to add two or more bits together it becomes slightly harder. In this case, we need to create a full adder circuits. The difference between a full adder and a half adder we looked at is that a full adder accepts inputs A and B plus a **carry-in** (CN-1) giving outputs Q and CN. Once we have a full adder, then we can string eight of them together to create a byte-wide adder and cascade the carry bit from one adder to the next. The logic table for a full adder is slightly more complicated than the tables we have used before, because now we have **3 input bits**. The truth table and the circuit diagram for a full-adder is shown in Fig. 2. If you look at the Q bit, it is 1 if an odd number of the three inputs is one, i.e., Q is the XOR of the three inputs. The full adder can be realized as shown below. Notice that the full adder can be constructed from two half adders and an ***OR*** gate.

One-bit Full Adder with Carry-In & Carry-Out

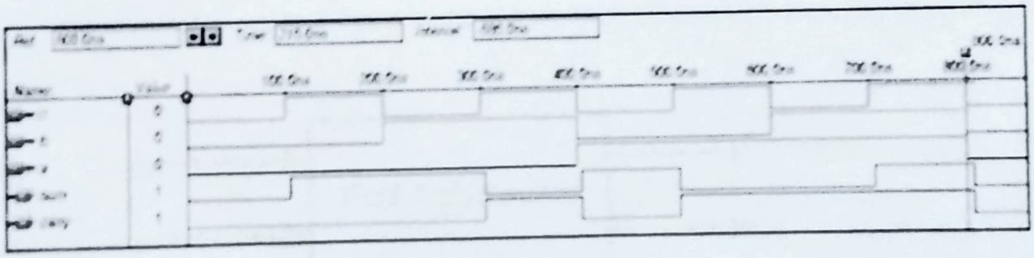
INPUTS			OUTPUT	
A	B	C-IN	C-OUT	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

VHDL Code for a Full Adder

```
Library ieee;
use ieee.std_logic_1164.all;

entity full_adder is port(a,b,c:in bit; sum,carry:out bit);
end full_adder;

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

→ can use any other formula for carry.

Waveforms



Tag: Half Subtractor and Full Subtractor PDF

Full Subtractor | Definition | Circuit Diagram | Truth Table

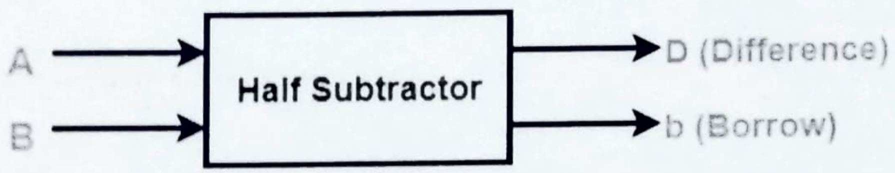
Digital Design

Half Subtractor-

Before you go through this article, make sure that you have gone through the previous article on **Half Subtractor**.

We have discussed-

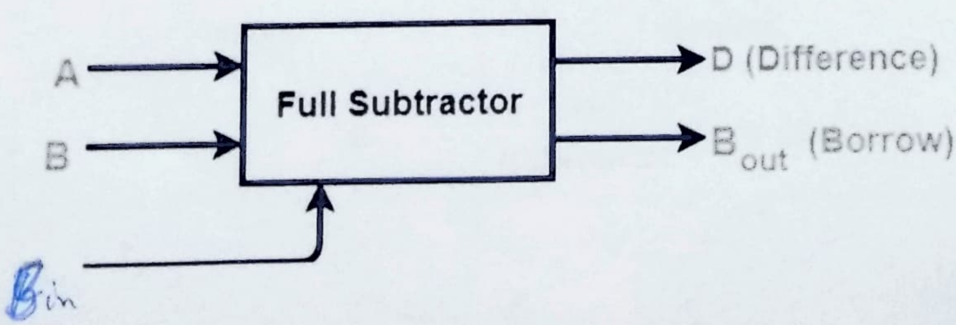
- Half Subtractor is used for the purpose of subtracting two single bit numbers.
- Half subtractors have no scope of taking into account "Borrow-in" from the previous circuit.
- To overcome this drawback, full subtractor comes into play.



In this article, we will discuss about Full Subtractor.

Full Subtractor-

- Full Subtractor is a combinational logic circuit.
- It is used for the purpose of subtracting two single bit numbers.
- It also takes into consideration borrow of the lower significant stage.
- Thus, full subtractor has the ability to perform the subtraction of three bits.
- Full subtractor contains 3 inputs and 2 outputs (Difference and Borrow) as shown-

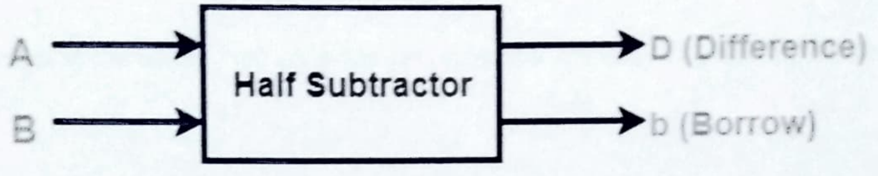


Half Subtractor | Definition | Circuit Diagram | Truth Table

Digital Design

Half Subtractor-

- Half Subtractor is a combinational logic circuit.
- It is used for the purpose of subtracting two single bit numbers.
- It contains 2 inputs and 2 outputs (difference and borrow).



Half Subtractor Designing-

Half subtractor is designed in the following steps-

Step-01:

Identify the input and output variables-

- Input variables = A, B (either 0 or 1)
- Output variables = D, b where D = Difference and b = borrow

Step-02:

Draw the truth table-

Inputs		Outputs	
A	B	D (Difference)	b (Borrow)
0	0	0	0

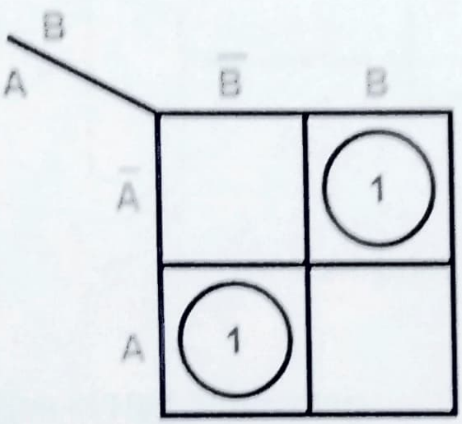
0	1	1	1
1	0	1	0
1	1	0	0

Truth Table

Step-03:

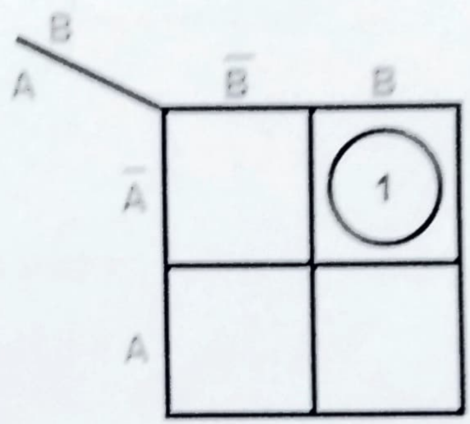
Draw K-maps using the above truth table and determine the simplified Boolean expressions

For D:



$D = A \oplus B$

For b:



$b = \overline{A}.B$

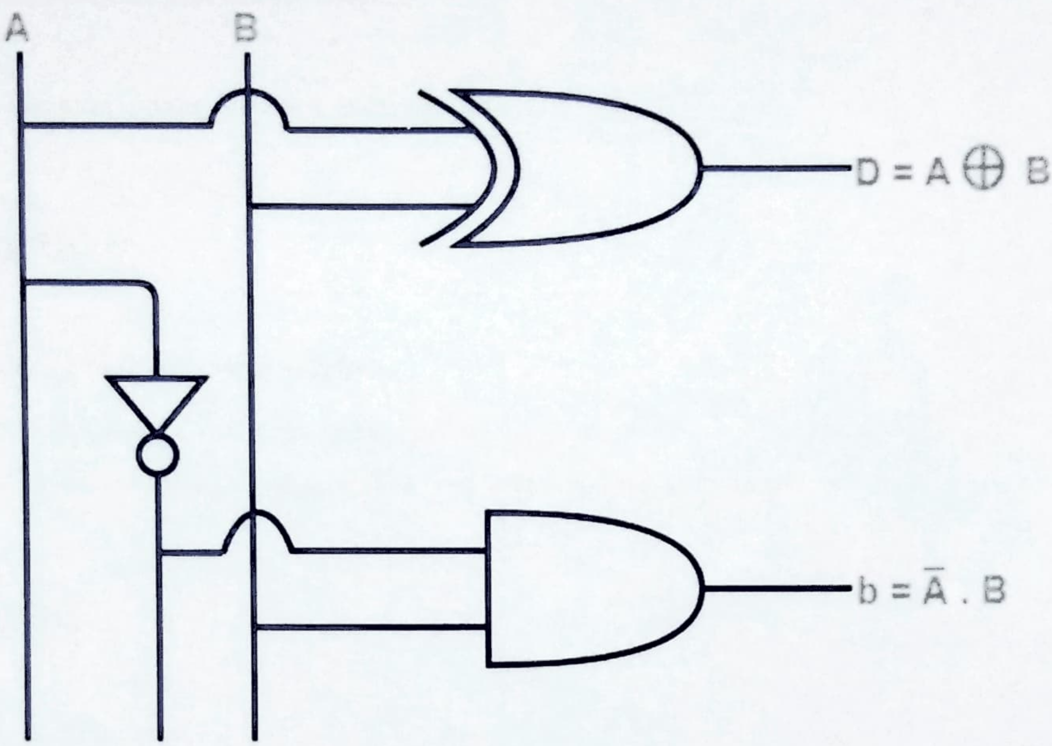
K Maps

Also Read- Half Adder

Step-04:

Draw the logic diagram.

The implementation of half subtractor using 1 XOR gate, 1 NOT gate and 1 AND gate is as shown below-



Half Subtractor Logic Diagram

Limitation of Half Subtractor-

- Half subtractors do not take into account "Borrow-in" from the previous circuit.
- This is a major drawback of half subtractors.
- This is because real time scenarios involve subtracting the multiple number of bits which can not be accomplished using half subtractors.

To overcome this drawback, Full Subtractor comes into play.

To gain better understanding about Half Subtractor,

Watch this Video Lecture

Also write code for behavioural model.

B_{in}

Designing a Full Subtractor-

Full subtractor is designed in the following steps-

Step-01:

Identify the input and output variables-

- Input variables = A, B, B_{in} (either 0 or 1)
- Output variables = D, B_{out} where D = Difference and B_{out} = Borrow

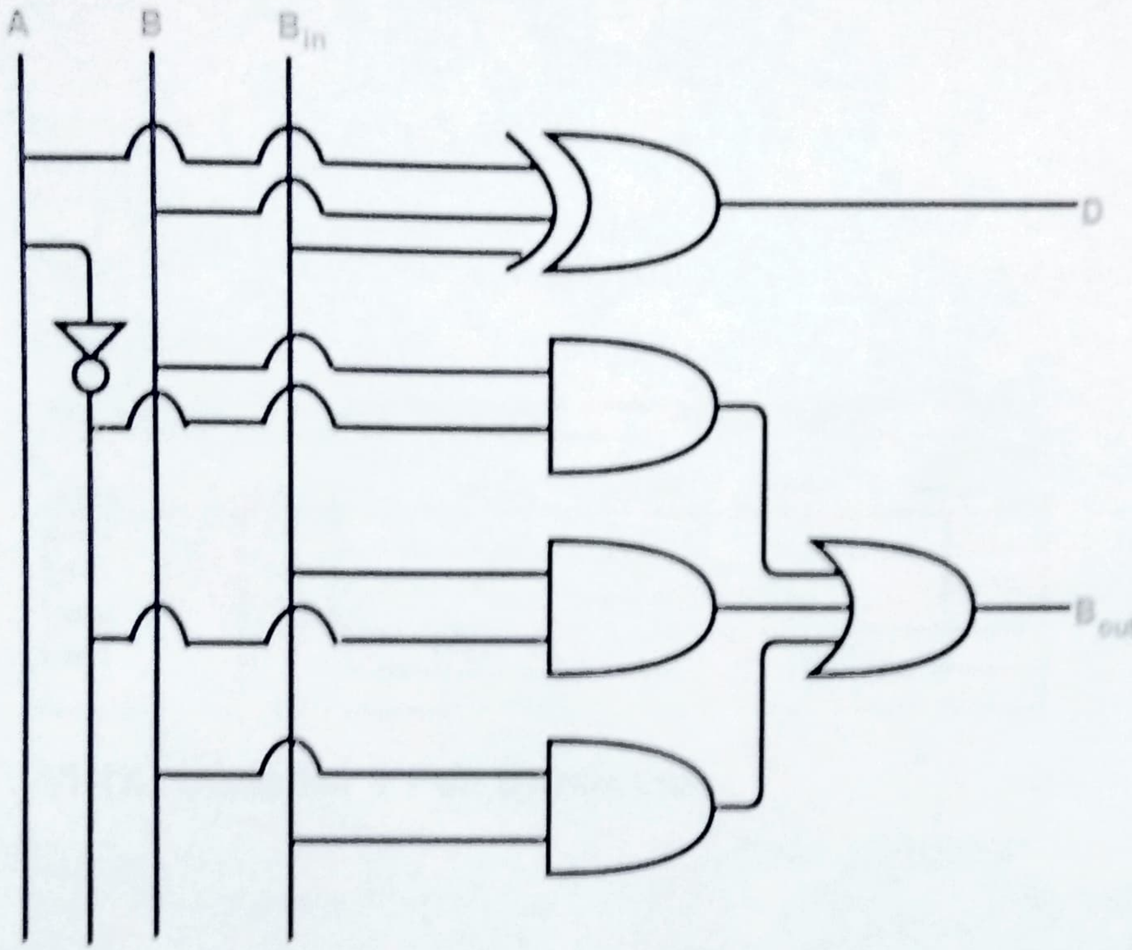
Step-02:

Draw the truth table-

Inputs			Outputs	
A	**B**	**B_{in}**	**B_{out} (Borrow)**	**D (Difference)**
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Draw the logic diagram.

The implementation of full adder using 1 XOR gate, 3 AND gates, 1 NOT gate and 1 OR gate is as shown below-



Full Subtractor Logic Diagram

To gain better understanding about Full Subtractor,

Watch this Video Lecture

Next Article- Ripple Carry Adder

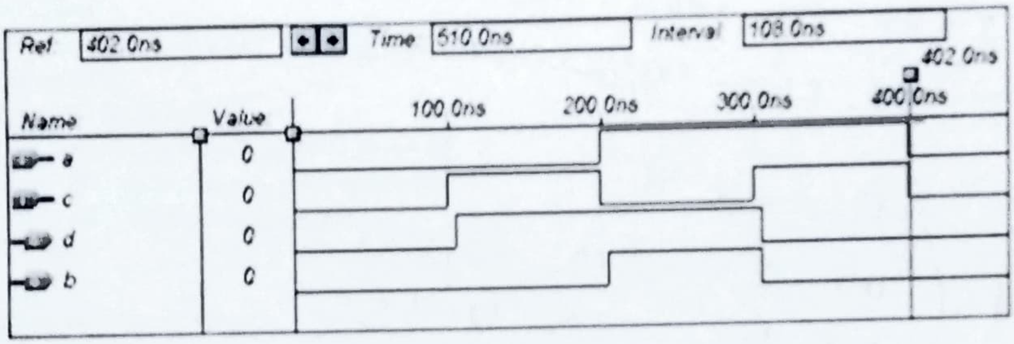
*Get more notes and other study material of **Digital Design***

*Watch video lectures by visiting our YouTube channel **LearnVidFun***

VHDL Code for a Half-Subtractor

```
Library ieee;
use ieee.std_logic_1164.all;

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

architecture data of half_sub is
begin
   d<= a xor c;
   b<= (a and (not c));
end data;
```

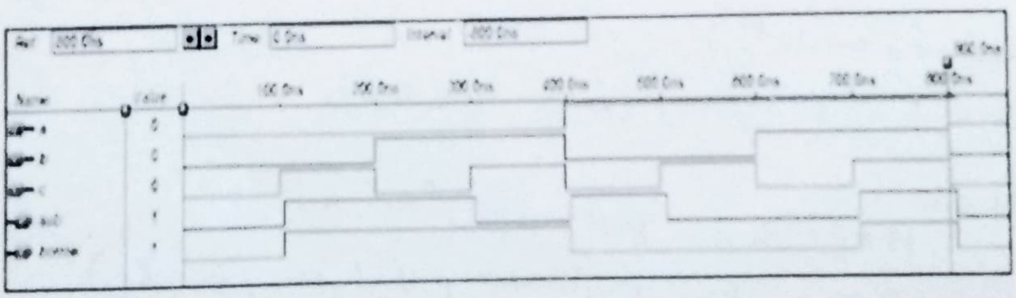
Waveforms



VHDL Code for a Full Subtractor

```
Library ieee;
use ieee.std_logic_1164.all;

entity full_sub is
   port(a,b,c:in bit; sub,borrow:out bit);
end full_sub;

architecture data of full_sub is
begin
   sub<= a xor b xor c;
   borrow <= ((b xor c) and (not a)) or (b and c);
end data;
```

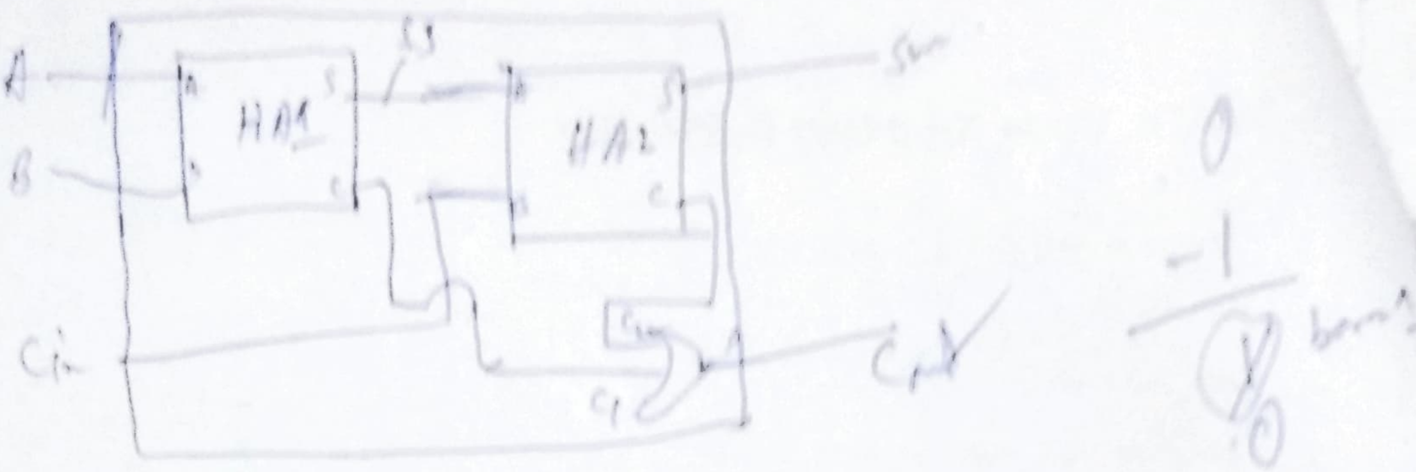
Waveforms



view source

print?

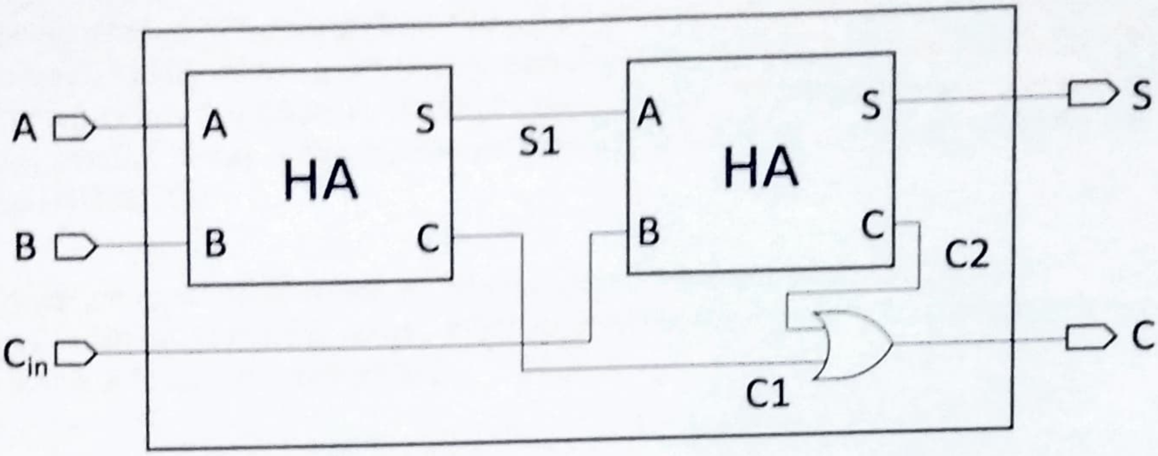
```
-- FULL ADDER
library ieee;
use ieee.std_logic_1164.all;
entity Full_Adder is
   port( X, Y, Cin : in std_logic;
         sum, Cout : out std_logic);
end Full_Adder;

architecture bhv of Full_Adder is
begin
   sum &amp;lt;= (X xor Y) xor Cin;
   Cout &amp;lt;= (X and (Y or Cin)) or (Cin and Y);
end bhv;
==============================================================
--4 bit Adder Subtractor
library ieee;
use ieee.std_logic_1164.all;
entity addsub is
   port( OP: in std_logic;
         A,B  : in std_logic_vector(3 downto 0);
         R  : out std_logic_vector(3 downto 0);
         Cout, OVERFLOW : out std_logic);
end addsub;

architecture struct of addsub is
component Full_Adder is
  port( X, Y, Cin : in std_logic;
        sum, Cout : out std_logic);
end component;
signal C1, C2, C3, C4: std_logic;
signal TMP: std_logic_vector(3 downto 0);

begin
TMP &lt;= A xor B;
FA0:Full_Adder port map(A(0),TMP(0),OP, R(0),C1);-- R0
FA1:Full_Adder port map(A(1),TMP(1),C1, R(1),C2);-- R1
FA2:Full_Adder port map(A(2),TMP(2),C2, R(2),C3);-- R2
FA3:Full_Adder port map(A(3),TMP(3),C3, R(3),C4);-- R3
OVERFLOW &lt;= C3 XOR C4 ;
Cout &lt;= C4;
end struct;
```



VHDL code

```
Library IEEE
use. IEEE.STD_logic_1164.ALL
use IEEE.STD_LOGIC_ARITH.ALL
use IEEE.STD_logic_UNSIGNED.ALL;

Entity full_add is
  port ( a: in STD_LOGIC;
         b: in STD_LOGIC;
         Cin: in STD_LOGIC;
         sum : out STD_logic;
         Cout : out STD_logic)
end full_add;
architecture Behavioral of full_add is
  Component ha is
    Port ( a: in STD_logic;
           b: in STD_logic;
           sha: out STD_logic;
           cha: out STD_logic;
  end component;
  signal s_s, c1, c2 : STD_logic;
  begin
    HA1: ha port map (a,b,s_s,c1);
    HA2: ha portmap (s_s,Cin,Sum,c2)
    Cout <= c1 or c2
end behavioral;
```

Full Adder using two Half Adders

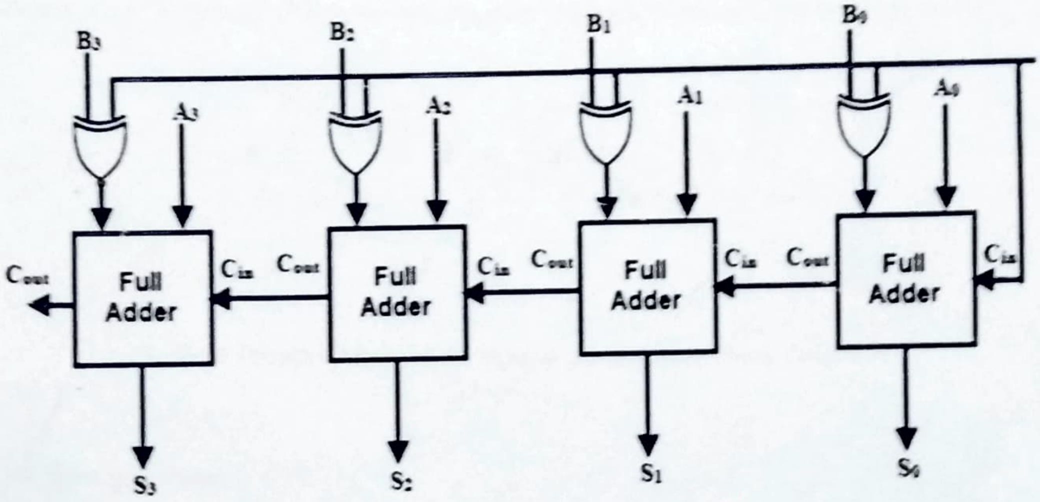



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity full_add is
   Port ( a : in  STD_LOGIC;
          b : in  STD_LOGIC;
          cin : in  STD_LOGIC;
          sum : out  STD_LOGIC;
          cout : out  STD_LOGIC);
end full_add;

architecture Behavioral of full_add is
   component ha is
 Port ( a : in  STD_LOGIC;
          b : in  STD_LOGIC;
          sha : out  STD_LOGIC;
          cha : out  STD_LOGIC);
end component;
signal s_s,c1,c2: STD_LOGIC ;
begin
HA1:ha port map(a,b,s_s,c1);
HA2:ha port map (s_s,cin,sum,c2);
cout<=c1 or c2 ;

end Behavioral;
```

Parallel Adder / Subtractor

The operations of both addition and subtraction can be performed by a one common binary adder. Such binary circuit can be designed by adding an Ex-OR gate with each full adder as shown in below figure. The figure below shows the 4 bit parallel binary adder/subtractor which has two 4 bit inputs as A3A2A1A0 and B3B2B1B0.

The mode input control line M is connected with carry input of the least significant bit of the full adder. This control line decides the type of operation, whether addition or subtraction.



When M= 1, the circuit is a subtractor and when M=0, the circuit becomes adder. The Ex-OR gate consists of two inputs to which one is connected to the B and other to input M. When M = 0, B Ex-OR of 0 produce B. Then full adders add the B with A with carry input zero and hence an addition operation is performed.

When M = 1, B Ex-OR of 0 produce B complement and also carry input is 1. Hence the complemented B inputs are added to A and 1 is added through the input carry, nothing but a 2's complement operation. Therefore, the subtraction operation is performed.

4. Parallel Adders

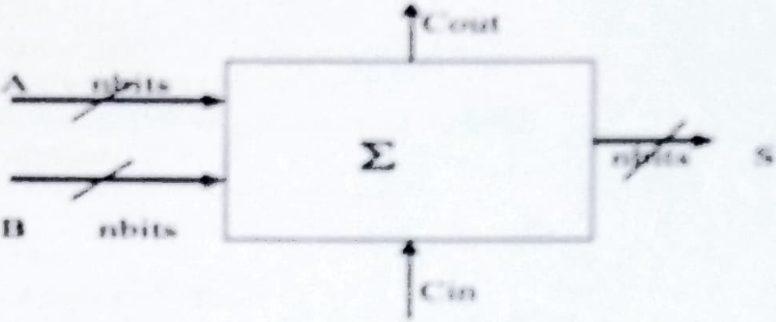


Fig. 3 Parallel Adder

Parallel adders are digital circuits that compute the addition of variable binary strings of equivalent or different size in parallel. The schematic diagram of a parallel adder is shown below in Fig. 3.

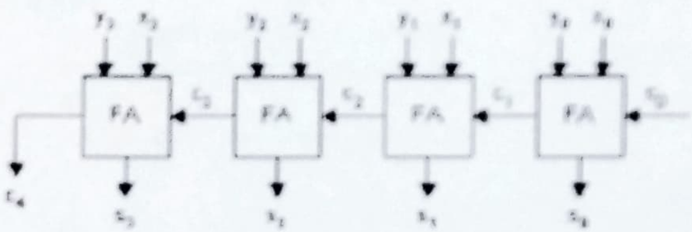


Figure 4: Parallel Adder: 4-bit Ripple-Carry Adder Block Diagram

4.1 Ripple-Carry adder

The ripple carry adder is constructed by cascading full adders (FA) blocks in series. One full adder is responsible for the addition of two binary digits at any stage of the ripple carry. The carryout of one stage is fed directly to the carry-in of the next stage. A number of full adders may be added to the ripple carry adder or ripple carry adders of different sizes may be cascaded in order to accommodate binary vector strings of larger sizes. For an n-bit parallel adder, it requires n computational elements (FA). Figure 4 shows an example of a parallel adder: a 4-bit ripple-carry adder. It is composed of four full adders. The augend's bits of x are added to the addend bits of y respectfully of their binary position. Each bit addition creates a sum and a carry out. The carry out is then transmitted to the carry in of the next higher-order bit. The final result creates a sum of four bits plus a carry out (c4). Even though this is a simple adder and can be used to add unrestricted bit length numbers, it is however not very efficient when large bit numbers are used. One of the most serious drawbacks of this adder is that the delay increases linearly with the bit length. As mentioned before, each full adder has to wait for the carry out of the previous stage to output steady-state result. Therefore even if the adder has a value at its output terminal, it has to wait for the propagation of the carry before the output reaches a correct value as shown in Fig. 5. Taking again the example in figure 4, the addition of x4 and y4 cannot reach steady state until c4 becomes available. In turn, c4 has to wait for c3, and so on down to c1.

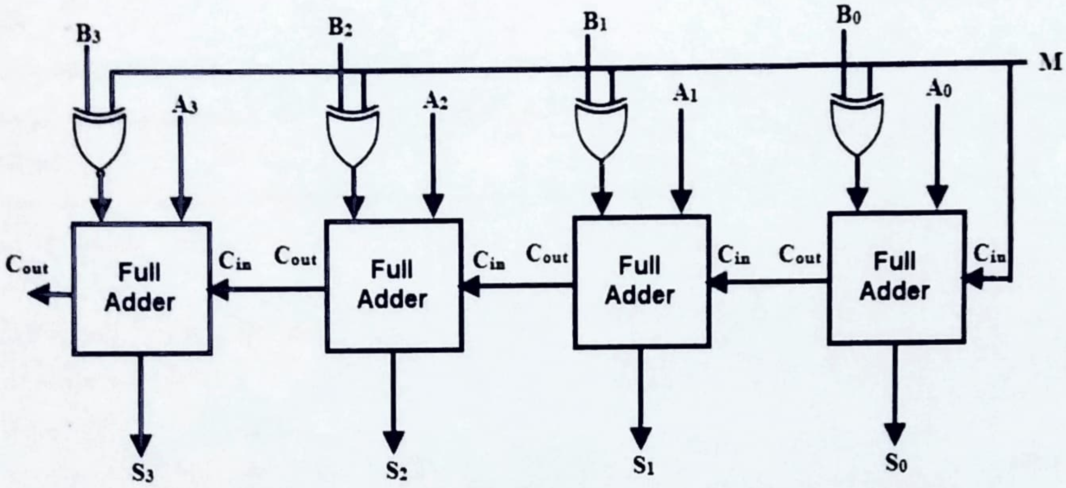
VHDL code for n-bit adder

```
-- function of adder:
-- A plus B to get n-bit sum and 1 bit carry
-- we may use generic statement to set the parameter
-- n of the adder.
-------------------------------------------------------

Library ieee;
Use  ieee.std_logic_1164.all;
Use  ieee.std_logic_arith.all;
Use ieee.std_logic_unsigned.all;

-------------------------------------------------------

entity ADDER is

generic(n: natural :=2);
port(      A:        in std_logic_vector(n-1 downto 0);
           B:        in std_logic_vector(n-1 downto 0);
           carry:    out std_logic;
           sum:      out std_logic_vector(n-1 downto 0)
);

end ADDER;

-------------------------------------------------------

Architecture behv of ADDER is

                          -- define a temp arary signal to store the result

signal result: std_logic_vector(n downto 0);

begin
                                  -- the 3rd bit should be carry

result<= ('0' & A) + ('0' & B);
sum<= result(n-1 downto 0);
carry<= result(n);

endbehv;

-------------------------------------------------------
```

Parallel Adder / Subtractor

The operations of both addition and subtraction can be performed by a one common binary adder. Such binary circuit can be designed by adding an Ex-OR gate with each full adder as shown in below figure. The figure below shows the 4 bit parallel binary adder/subtractor which has two 4 bit inputs as A3A2A1A0 and B3B2B1B0.

The mode input control line M is connected with carry input of the least significant bit of the full adder. This control line decides the type of operation, whether addition or subtraction.



When M= 1, the circuit is a subtractor and when M=0, the circuit becomes adder. The Ex-OR gate consists of two inputs to which one is connected to the B and other to input M. When M = 0, B Ex-OR of 0 produce B. Then full adders add the B with A with carry input zero and hence an addition operation is performed.

When M = 1, B Ex-OR of 0 produce B complement and also carry input is 1. Hence the complemented B inputs are added to A and 1 is added through the input carry, nothing but a 2's complement operation. Therefore, the subtraction operation is performed.

view source

print?

```
-- FULL ADDER
library ieee;
use ieee.std_logic_1164.all;
entity Full_Adder is
   port( X, Y, Cin : in std_logic;
         sum, Cout : out std_logic);
end Full_Adder;

architecture bhv of Full_Adder is
begin
   sum &amp;lt;= (X xor Y) xor Cin;
   Cout &amp;lt;= (X and (Y or Cin)) or (Cin and Y);
end bhv;
---------------------------------------------------------
--4 bit Adder Subtractor
library ieee;
use ieee.std_logic_1164.all;
entity addsub is
   port( OP: in std_logic;
         A,B  : in std_logic_vector(3 downto 0);
         R  : out std_logic_vector(3 downto 0);
         Cout, OVERFLOW : out std_logic);
end addsub;

architecture struct of addsub is
component Full_Adder is
  port( X, Y, Cin : in std_logic;
        sum, Cout : out std_logic);
end component;
signal C1, C2, C3, C4: std_logic;
signal TMP: std_logic_vector(3 downto 0);

begin
TMP &lt;= A xor B;
FA0:Full_Adder port map(A(0),TMP(0),OP, R(0),C1);-- R0
FA1:Full_Adder port map(A(1),TMP(1),C1, R(1),C2);-- R1
FA2:Full_Adder port map(A(2),TMP(2),C2, R(2),C3);-- R2
FA3:Full_Adder port map(A(3),TMP(3),C3, R(3),C4);-- R3
```

```
OVERFLOW &lt;= C3 XOR C4 ;
Cout &lt;= C4;
end struct;
```

Carry Look Ahead Adder | 4-bit Carry Look Ahead Adder

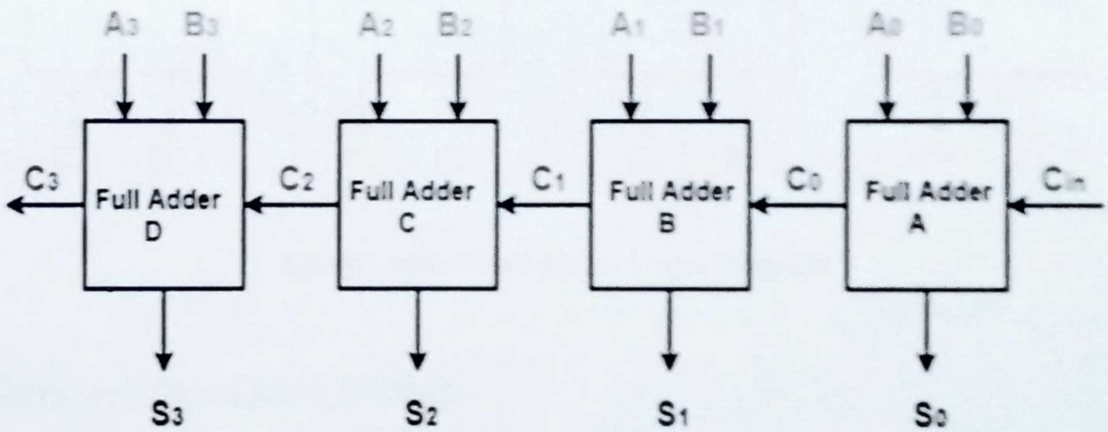
Digital Design

Ripple Carry Adder-

Before you go through this article, make sure that you have gone through the previous article on Ripple Carry Adder.

In Ripple Carry Adder,

- Each full adder has to wait for its carry-in from its previous stage full adder.
- Thus, n^{th} full adder has to wait until all (n-1) full adders have completed their operations.
- This causes a delay and makes ripple carry adder extremely slow.
- The situation becomes worst when the value of n becomes very large.
- To overcome this disadvantage, Carry Look Ahead Adder comes into play.



4-bit Ripple Carry Adder

In this article, we will discuss about Carry Look Ahead Adder.

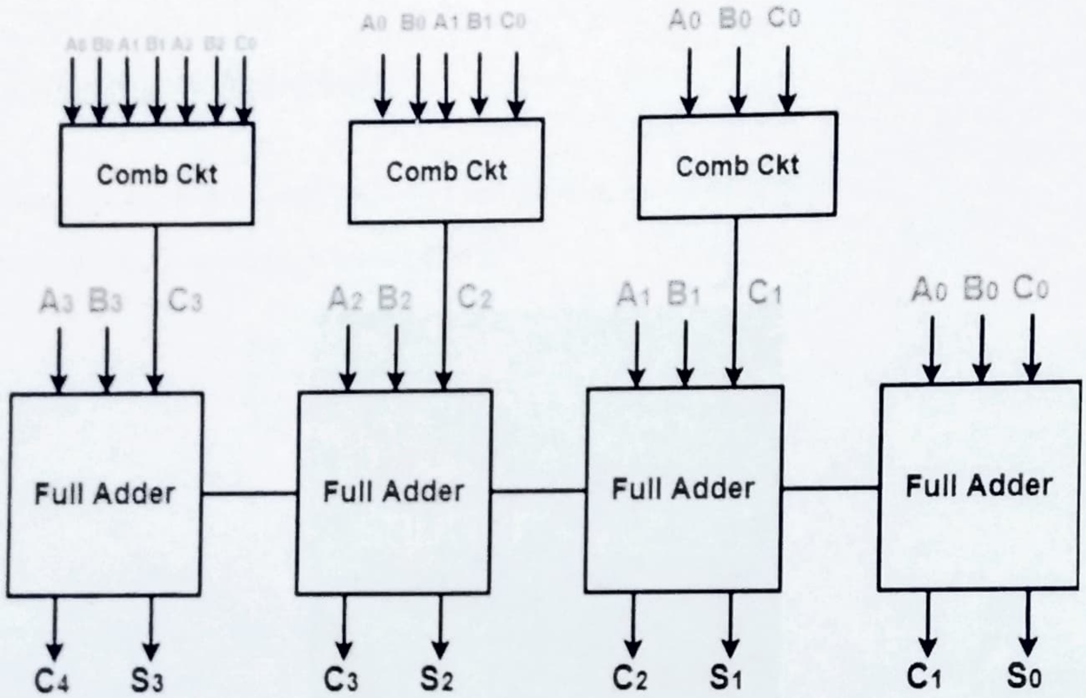
Carry Look Ahead Adder-

- Carry Look Ahead Adder is an improved version of the ripple carry adder.

- It generates the carry-in of each full adder simultaneously without causing any delay.
- The time complexity of carry look ahead adder = Θ (logn).

Logic Diagram-

The logic diagram for carry look ahead adder is as shown below-



Carry Look Ahead Adder Logic Diagram

Carry Look Ahead Adder Working-

> The working of carry look ahead adder is based on the principle-
>
> The carry-in of any stage full adder is independent of the carry bits generated during intermediate stages.

The carry-in of any stage full adder depends only on the following two parameters-

- Bits being added in the previous stages
- Carry-in provided in the beginning

Now,

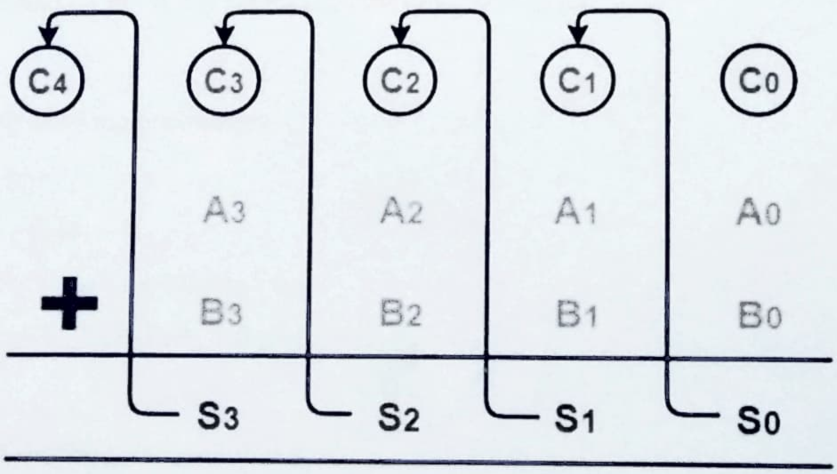
- The above two parameters are always known from the beginning.
- So, the carry-in of any stage full adder can be evaluated at any instant of time.
- Thus, any full adder need not wait until its carry-in is generated by its previous stage full adder.

Also Read- <u>Full Adder Working</u>

4-Bit Carry Look Ahead Adder-

Consider two 4-bit binary numbers $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$ are to be added.

Mathematically, the two numbers will be added as-



Adding two 4-bit Numbers

From here, we have-

$$C_1 = C_0 (A_0 \oplus B_0) + A_0B_0$$

$$C_2 = C_1 (A_1 \oplus B_1) + A_1B_1$$

$$C_3 = C_2 (A_2 \oplus B_2) + A_2B_2$$

$$C_4 = C_3 (A_3 \oplus B_3) + A_3B_3$$

For simplicity, Let-

- $G_i = A_iB_i$ where G is called carry generator
- $P_i = A_i \oplus B_i$ where P is called carry propagator

Then, re-writing the above equations, we have-

$$C_1 = C_0P_0 + G_0 \quad \ldots\ldots\ldots\ldots (1)$$

$$C_2 = C_1P_1 + G_1 \quad \ldots\ldots\ldots\ldots (2)$$

$$C_3 = C_2P_2 + G_2 \quad \ldots\ldots\ldots\ldots (3)$$

$$C_4 = C_3P_3 + G_3 \quad \ldots\ldots\ldots\ldots (4)$$

Now,

- Clearly, C1, C2 and C3 are intermediate carry bits.
- So, let's remove C_1, C_2 and C_3 from RHS of every equation.
- Substituting (1) in (2), we get C_2 in terms of C_0.
- Then, substituting (2) in (3), we get C_3 in terms of C_0 and so on.

Finally, we have the following equations-

- $C_1 = C_0P_0 + G_0$
- $C_2 = C_0P_0P_1 + G_0P_1 + G_1$
- $C_3 = C_0P_0P_1P_2 + G_0P_1P_2 + G_1P_2 + G_2$
- $C_4 = C_0P_0P_1P_2P3 + G_0P_1P_2P_3 + G_1P_2P_3 + G_2P_3 + G_3$

These equations are important to remember.

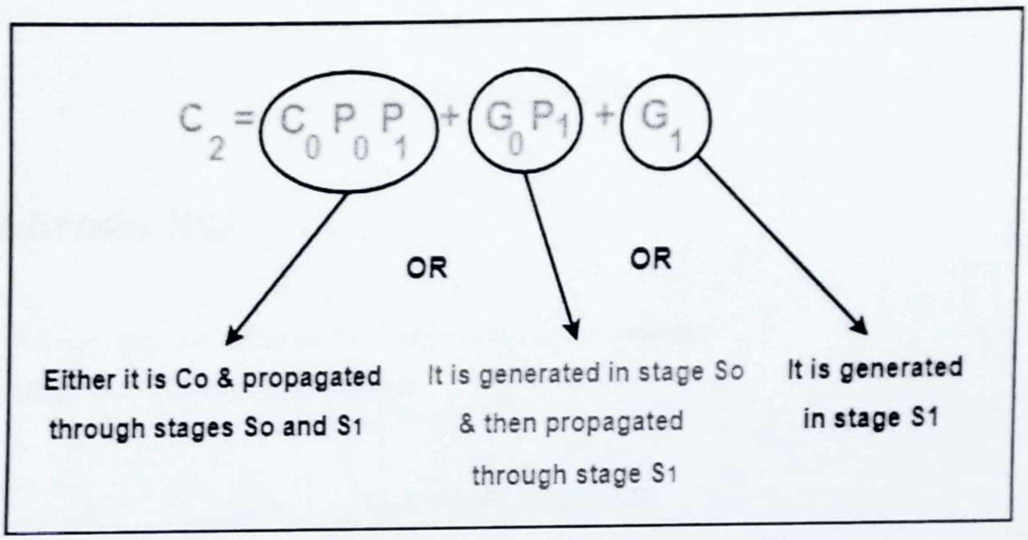
These equations show that the carry-in of any stage full adder depends only on-

- Bits being added in the previous stages
- Carry bit which was provided in the beginning

Trick To Memorize Above Equations-

As an example, let us consider the equation for generating carry bit C_2.

There are three possible reasons for generation of C_2 as depicted in the following picture-



In the similar manner, we can write other equations as well very easily.

Implementation Of Carry Generator Circuits-

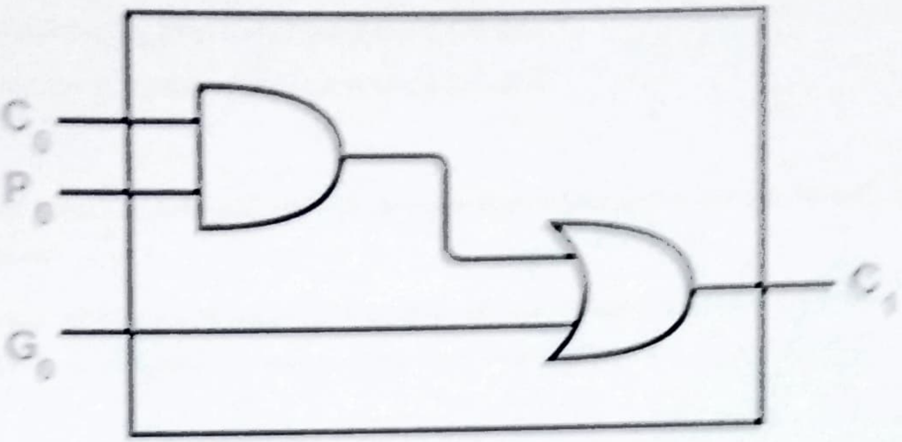
The above carry generator circuits are usually implemented as-

- Two level combinational circuits.
- Using AND and OR gates where gates are assumed to have any number of inputs.

Implementation Of C_1-

- The carry generator circuit for C1 is implemented as shown below.
- It requires 1 AND gate and 1 OR gate.

$$C_1 = C_0P_0 + G_0$$

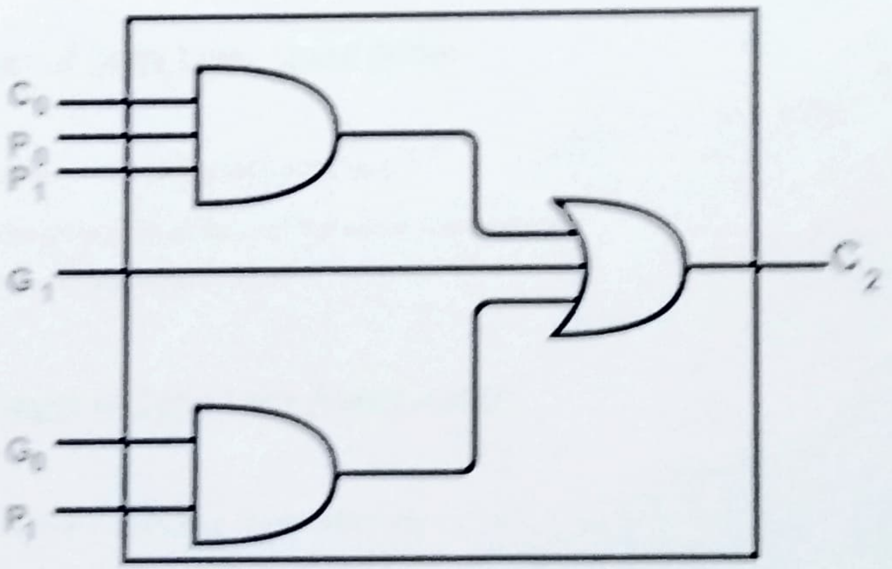


Implementation of C1

Implementation Of C_2-

- The carry generator circuit for C2 is implemented as shown below.
- It requires 2 AND gates and 1 OR gate.

$$C_2 = C_0P_0P_1 + G_0P_1 + G_1$$



Implementation of C2

Implementation Of C_3 & C_4-

Similarly, we implement C_3 and C_4.

- Implementation of C_3 uses 3 AND gates and 1 OR gate.
- Implementation of C_4 uses 4 AND gates and 1 OR gate.

Total number of gates required to implement carry generators (provided carry propagators P_i and carry generators G_i) are-

- Total number of AND gates required for addition of 4-bit numbers = 1 + 2 + 3 + 4 = 10.
- Total number of OR gates required for addition of 4-bit numbers = 1 + 1 + 1 + 1 = 4.

General Formula-

The following formula is used to calculate number of gates required for evaluating all carry bits-

For a n-bit carry look ahead adder to evaluate all the carry bits, it requires-

- Number of AND gates = n(n+1) / 2
- Number of OR gates = n

Advantages of Carry Look Ahead Adder-

The advantages of carry look ahead adder are-

- It generates the carry-in for each full adder simultaneously.
- It reduces the propagation delay.

Disadvantages of Carry Look Ahead Adder-

The disadvantages of carry look ahead adder are-

- It involves complex hardware.
- It is costlier since it involves complex hardware.
- It gets more complicated as the number of bits increases.

To gain better understanding about Carry Look Ahead Adder,

VHDL Code for Partial Full Adder

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

entity Partial_Full_Adder is
Port ( A : in STD_LOGIC;
B : in STD_LOGIC;
Cin : in STD_LOGIC;
S : out STD_LOGIC;
P : out STD_LOGIC;
G : out STD_LOGIC);
end Partial_Full_Adder;

architecture Behavioral of Partial_Full_Adder is

begin

S <= A xor B xor Cin;
P <= A xor B;
G <= A and B;

end Behavioral;
```

VHDL Code for Carry Look Ahead Adder

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

entity Carry_Look_Ahead is
Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
B : in STD_LOGIC_VECTOR (3 downto 0);
Cin : in STD_LOGIC;
S : out STD_LOGIC_VECTOR (3 downto 0);
Cout : out STD_LOGIC);
end Carry_Look_Ahead;

architecture Behavioral of Carry_Look_Ahead is

component Partial_Full_Adder
Port ( A : in STD_LOGIC;
B : in STD_LOGIC;
Cin : in STD_LOGIC;
S : out STD_LOGIC;
P : out STD_LOGIC;
G : out STD_LOGIC);
end component;

signal c1,c2,c3: STD_LOGIC;
signal P,G: STD_LOGIC_VECTOR(3 downto 0);
begin

PFA1: Partial_Full_Adder port map( A(0), B(0), Cin, S(0), P(0),
G(0));
```

```
PFA2: Partial_Full_Adder port map( A(1), B(1), c1, S(1), P(1),
G(1));
PFA3: Partial_Full_Adder port map( A(2), B(2), c2, S(2), P(2),
G(2));
PFA4: Partial_Full_Adder port map( A(3), B(3), c3, S(3), P(3),
G(3));

c1 <= G(0) OR (P(0) AND Cin);
c2 <= G(1) OR (P(1) AND G(0)) OR (P(1) AND P(0) AND Cin);
c3 <= G(2) OR (P(2) AND G(1)) OR (P(2) AND P(1) AND G(0)) OR
(P(2) AND P(1) AND P(0) AND Cin);
Cout <= G(3) OR (P(3) AND G(2)) OR (P(3) AND P(2) AND G(1)) OR
(P(3) AND P(2) AND P(1) AND G(0)) OR (P(3) AND P(2) AND P(1) AND
P(0) AND Cin);

end Behavioral;
```

VHDL Testbench Code for Carry Look Ahead Adder

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY Tb_Carry_Look_Ahead IS
END Tb_Carry_Look_Ahead;

ARCHITECTURE behavior OF Tb_Carry_Look_Ahead IS

-- Component Declaration for the Unit Under Test (UUT)

COMPONENT Carry_Look_Ahead
PORT(
A : IN std_logic_vector(3 downto 0);
B : IN std_logic_vector(3 downto 0);
Cin : IN std_logic;
S : OUT std_logic_vector(3 downto 0);
Cout : OUT std_logic
);
END COMPONENT;

--Inputs
signal A : std_logic_vector(3 downto 0) := (others => '0');
signal B : std_logic_vector(3 downto 0) := (others => '0');
signal Cin : std_logic := '0';

--Outputs
signal S : std_logic_vector(3 downto 0);
signal Cout : std_logic;

BEGIN

-- Instantiate the Unit Under Test (UUT)
uut: Carry_Look_Ahead PORT MAP (
A => A,
B => B,
Cin => Cin,
```