Carry Look Ahead Adders

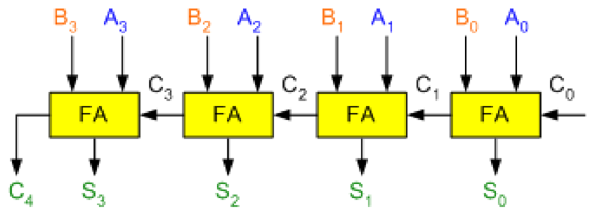
Lesson Objectives:

The objectives of this lesson are to learn about:

1. Carry Look Ahead Adder circuit.
2. Binary Parallel Adder/Subtractor circuit.
3. BCD adder circuit.
4. Binary mutiplier circuit.

Carry Look Ahead Adder:

In ***ripple carry adders***, the carry propagation time is the major speed limiting factor as seen in the previous lesson.



Most other arithmetic operations, e.g. multiplication and division are implemented using several add/subtract steps. Thus, improving the speed of addition will improve the speed of all other arithmetic operations.

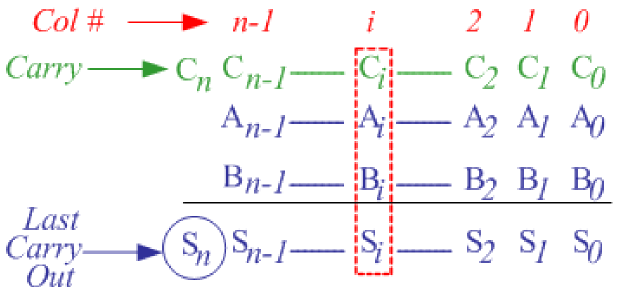
Accordingly, reducing the carry propagation delay of adders is of great importance. Different logic design approaches have been employed to overcome the carry propagation problem.

One widely used approach employs the principle of ***carry look-ahead*** solves this problem by calculating the carry signals in advance, based on the input signals.

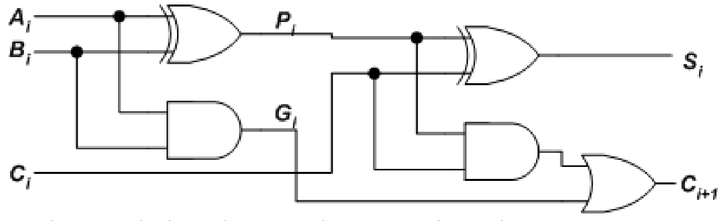
This type of adder circuit is called as ***carry look-ahead adder (CLA adder).*** It is based on the fact that a carry signal will be generated in two cases:

(1) when both bits A_i and B_i are 1, or
(2) when one of the two bits is 1 and the carry-in (carry of the previous stage) is 1.

To understand the carry propagation problem, let's consider the case of adding two *n-bit* numbers ***A*** and ***B***.


Col #
$n-1$
i
2
1
0
Carry
C_n C_{n-1} — C_i — C_2 C_1 C_0
A_{n-1} — A_i — A_2 A_1 A_0
B_{n-1} — B_i — B_2 B_1 B_0
Last Carry Out
S_n S_{n-1} — S_i — S_2 S_1 S_0

The Figure shows the full adder circuit used to add the operand bits in the i^{th} column; *namely A_i & B_i and the carry bit coming from the previous column (C_i).*



In this circuit, the 2 internal signals P_i and G_i are given by:

$$P_i = A_i \oplus B_i \text{ ....}\textbf{(1)}$$

$$G_i = A_i B_i \text{ ....}\textbf{(2)}$$

The output sum and carry can be defined as :

$$S_i = P_i \oplus C_i \text{ ....}\textbf{(3)}$$

$$C_{i+1} = G_i + P_i C_i \text{ ....}\textbf{(4)}$$

G_i is known as the ***carry Generate*** signal since a carry (C_{i+1}) is generated whenever G_i =1, regardless of the input carry (C_i).

P_i is known as the ***carry propagate*** signal since whenever P_i =1, the input carry is propagated to the output carry, i.e., C_{i+1}. = C_i (note that whenever P_i =1, G_i =0).

Computing the values of P_i and G_i only depend on the input operand bits (A_i & B_i) as clear from the Figure and equations.

Thus, these signals settle to their ***steady-state value*** after the propagation through their respective gates.

Computed values of ***all*** the P_i's are valid one XOR-gate delay after the operands A and B are made valid.

Computed values of ***all*** the G_i's are valid one AND-gate delay after the operands A and B are made valid.

The Boolean expression of the carry outputs of various stages can be written as follows:

$\mathbf{C_1 = G_0 + P_0C_0}$
$\mathbf{C_2 = G_1 + P_1C_1 = G_1 + P_1 (G_0 + P_0C_0)}$
$\mathbf{= G_1 + P_1G_0 + P_1P_0C_0}$
$\mathbf{C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0}$
$\mathbf{C_4 = G_3 + P_3C_3}$
$\mathbf{= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0}$

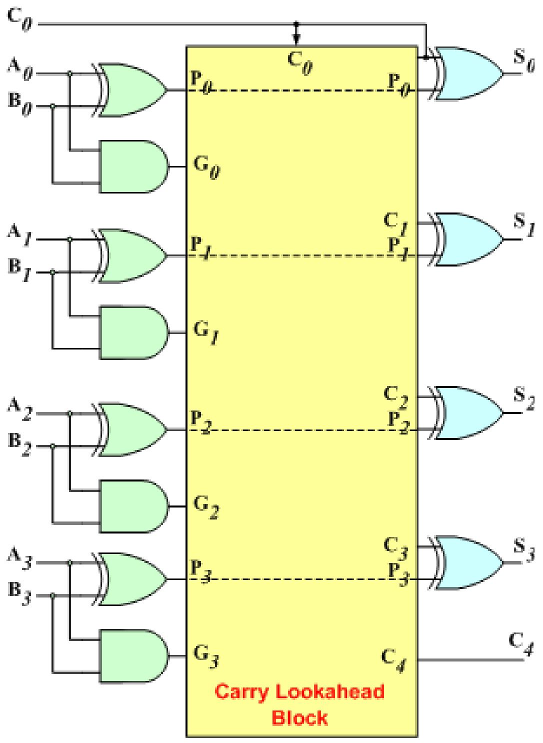
In general, the *i*th. carry output is expressed in the form **C_i** = F_i (P's, G's , **C_0**).

In other words, each carry signal is expressed as a direct SOP function of **C_0** rather than its preceding carry signal.

Since the Boolean expression for each output carry is expressed in SOP form, it can be implemented in two-level circuits.

The 2-level implementation of the carry signals has a propagation delay of 2 gates, i.e., 2τ.

The 4-bit carry look-ahead (CLA) adder consists of 3 levels of logic:



First level: Generates all the P & G signals. Four sets of P & G logic (each consists of an XOR gate and an AND gate). Output signals of this level (P's & G's) will be valid after 1τ.

Second level: The Carry Look-Ahead (CLA) logic block which consists of four 2-level implementation logic circuits. It generates the carry signals (C_1, C_2, C_3, and C_4) as defined by the above expressions. Output signals of this level (C_1, C_2, C_3, and C_4) will be valid after 3τ.

Third level: Four XOR gates which generate the sum signals (S_i) ($S_i = P_i \oplus C_i$). Output signals of this level (S_0, S_1, S_2, and S_3) will be valid after 4τ.

Thus, the 4 Sum signals (S_0, S_1, S_2 & S_3) will all be valid after a total delay of 4τ compared to a delay of $(2n+1)\tau$ for Ripple Carry adders.

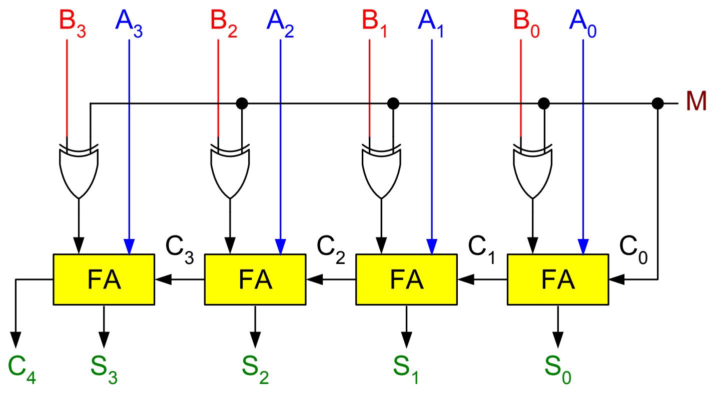
For a 4-bit adder ($n = 4$), the Ripple Carry adder delay is 9τ.

The disadvantage of the CLA adders is that the carry expressions (and hence logic) become quite complex for more than 4 bits.

Thus, CLA adders are usually implemented as 4-bit modules that are used to build larger size adders.

Binary Parallel Adder/Subtractor:

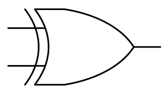
The addition and subtraction operations can be done using an Adder-Subtractor circuit. The figure shows the logic diagram of a 4-bit Adder-Subtractor circuit.



The circuit has a mode control signal M which determines if the circuit is to operate as an adder or a subtractor.

Each XOR gate receives input M and one of the inputs of B, i.e., B_i. To understand the behavior of XOR gate consider its truth table given below. If one input of XOR gate is **zero** then the output of XOR will be **same** as the second input. While if one input of XOR gate is **one** then the output of XOR will be **complement** of the second input.

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0



(see animation in authorware)

So when **M = 0**, the output of XOR gate will be $B_i \oplus 0 = B_i$. If the full adders receive the value of B, and the input carry C_0 is 0, the circuit performs **A plus B**.

When **M = 1**, the output of XOR gate will be $B_i \oplus 1 = B_i'$. If the full adders receive the value of B', and the input carry C_0 is 1, the circuit performs A plus 1's complement of B plus 1, which is equal to **A minus B**.

BCD Adder:

If two BCD digits are added then their sum result will not always be in BCD.
Consider the two given examples.

Correct: Result is in BCD.

$$\begin{array}{rcr} 0110 & = & 6 \\ +\underline{0011} & = & +\underline{3} \\ \underline{1001} & = & \underline{9} \end{array}$$

Wrong: Result is not in BCD.

$$\begin{array}{rcr} 0101 & = & 5 \\ +\underline{0111} & = & +\underline{7} \\ \underline{1100} & = & \underline{12} \end{array}$$

In the first example, result is in BCD while in the second example it is not in BCD.

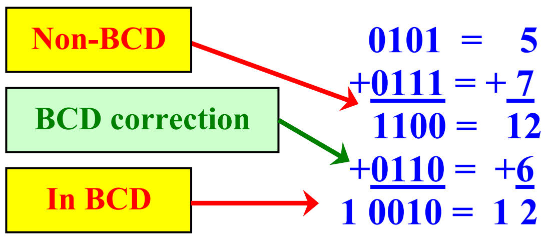
Four bits are needed to represent all BCD digits (0 – 9). But with four bits we can represent up to 16 values (0000 through 1111). The extra six values (1010 through 1111) are **not valid** BCD digits.

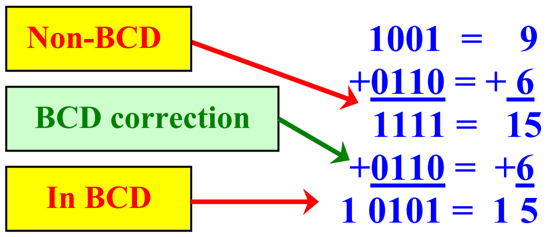
Whenever the sum result is > **9**, it will not be in BCD and will require correction to get a valid BCD result.

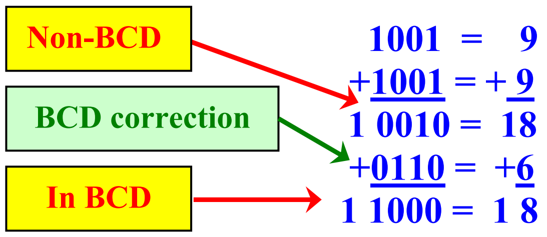
Z_3	Z_2	Z_1	Z_0		F
0	0	0	0		0
0	0	0	1		0
0	0	1	0		0
0	0	1	1		0
0	1	0	0		0
0	1	0	1		0
0	1	1	0		0
0	1	1	1		0
1	0	0	0		0
1	0	0	1		0
1	0	1	0		1
1	0	1	1		1
1	1	0	0		1
1	1	0	1		1
1	1	1	0		1
1	1	1	1		1

Correction is done through the addition of 6 to the result to skip the six invalid values as shown in the truth table by **yellow** color.

Consider the given examples of non-BCD sum result and its correction.







A BCD adder is a circuit that adds two BCD digits in parallel and produces a sum BCD digit and a carry out bit.

The maximum sum result of a *BCD input* adder can be 19. As maximum number in BCD is 9 and may be there will be a carry from previous stage also, so 9 + 9 + 1 = 19

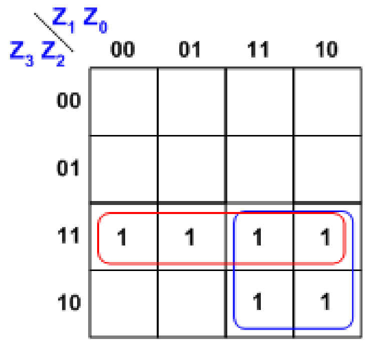
The following truth table shows all the possible sum results when two BCD digits are added.

Dec	CO	Z_3	Z_2	Z_1	Z_0		F
0	0	0	0	0	0		0
1	0	0	0	0	1		0
2	0	0	0	1	0		0
3	0	0	0	1	1		0
4	0	0	1	0	0		0
5	0	0	1	0	1		0
6	0	0	1	1	0		0
7	0	0	1	1	1		0
8	0	1	0	0	0		0
9	0	1	0	0	1		0
10	0	1	0	1	0		1
11	0	1	0	1	1		1
12	0	1	1	0	0		1
13	0	1	1	0	1		1
14	0	1	1	1	0		1
15	0	1	1	1	1		1
16	1	0	0	0	0		1
17	1	0	0	0	1		1
18	1	0	0	1	0		1
19	1	0	0	1	1		1

The logic circuit that checks the necessary BCD correction can be derived by detecting the condition where the resulting binary sum is 01010 through 10011 (decimal 10 through 19).

It can be done by considering the shown truth table, in which the function ***F*** is true when the digit is not a valid BCD digit. It can be simplified using a 5-variable K-map.

But detecting values 1010 through 1111 (decimal 10 through 15) can also be done by using a 4-variable K-map as shown in the figure.

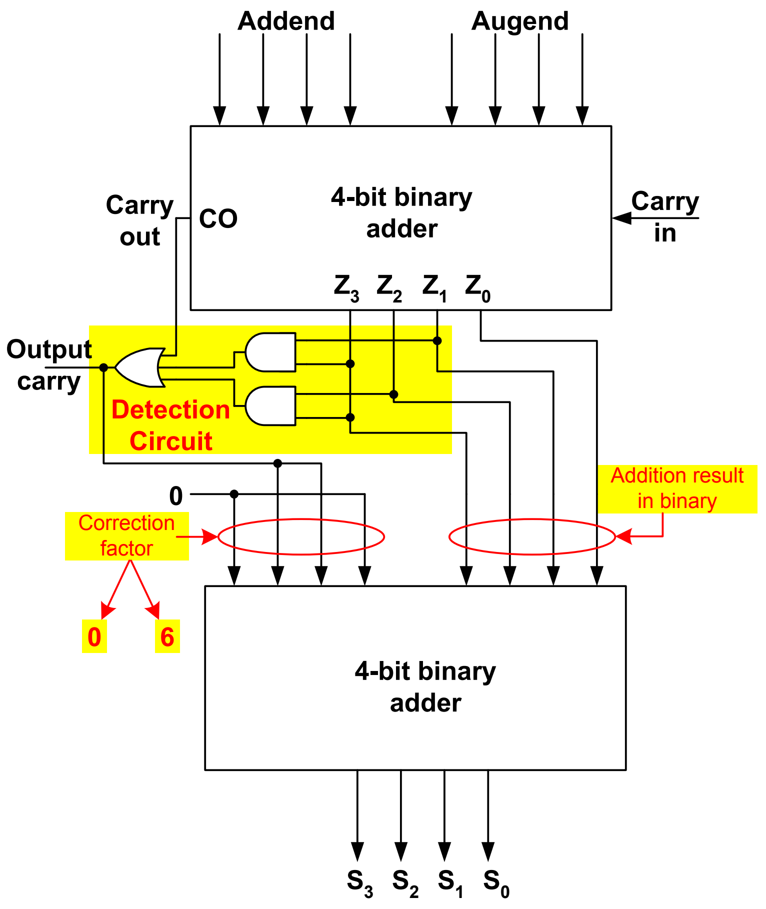


$F = Z_3 Z_2 + Z_3 Z_1$

Values greater than 1111, i.e., from 10000 through 10011 (decimal 16 through 19) can be detected by the carry out (CO) which equals 1 only for these output values. So, F = CO = 1 for these values. Hence, *F* is true when *CO* is true **OR** when **(**$Z_3 Z_2 + Z_3 Z_1$**)** is true.
Thus, the correction step (adding 0110) is performed if the following function equals 1:

$$F = CO + Z_3 Z_2 + Z_3 Z_1$$

The circuit of the BCD adder will be as shown in the figure.



The two BCD digits, together with the input carry, are first added in the top 4-bit binary adder to produce the binary sum. The bottom 4-bit binary adder is used to add the correction factor to the binary result of the top binary adder.

Note:

- When the **Output carry** is equal to **zero**, the correction factor equals zero.
- When the **Output carry** is equal to **one**, the correction factor is 0110.

The output carry generated from the bottom binary adder is ignored, since it supplies information already available at the **output-carry** terminal.

A decimal parallel adder that adds **n** decimal digits needs **n** BCD adder stages. The output carry from one stage must be connected to the input carry of the next higher-order stage.

Binary Multiplier:

Multiplication of binary numbers is performed in the same way as with decimal numbers.

The multiplicand is multiplied by each bit of the multiplier, starting from the least significant bit.

The result of each such multiplication forms a partial product. Successive partial products are shifted one bit to the left.

The product is obtained by adding these shifted partial products.

Example 1: Consider an example of multiplication of two numbers, say A and B (2 bits each), ***C* = *A* x *B*.**

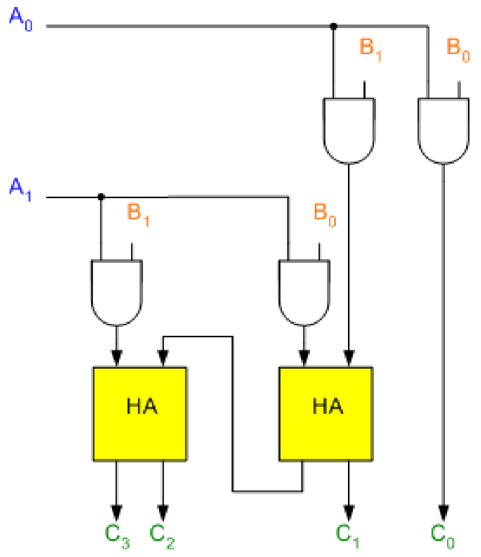
The first partial product is formed by multiplying the B_1B_0 by A_0. The multiplication of two bits such as A_0 and B_0 produces a 1 if both bits are 1; otherwise it produces a 0 like an AND operation. So the partial products can be implemented with AND gates.

The second partial product is formed by multiplying the B_1B_0 by A_1 and is shifted one position to the left.

$$\begin{array}{rrrr}
 & & B_1 & B_0 \\
 & \times & A_1 & A_0 \\
\hline
 & & A_0B_1 & A_0B_0 \\
 & A_1B_1 & A_1B_0 & \\
\hline
C_3 & C_2 & C_1 & C_0
\end{array}$$

(see animation in authorware)

The two partial products are added with two half adders (HA). Usually there are more bits in the partial products, and then it will be necessary to use FAs.



The least significant bit of the product does not have to go through an adder, since it is formed by the output of the first AND gate as shown in the Figure.

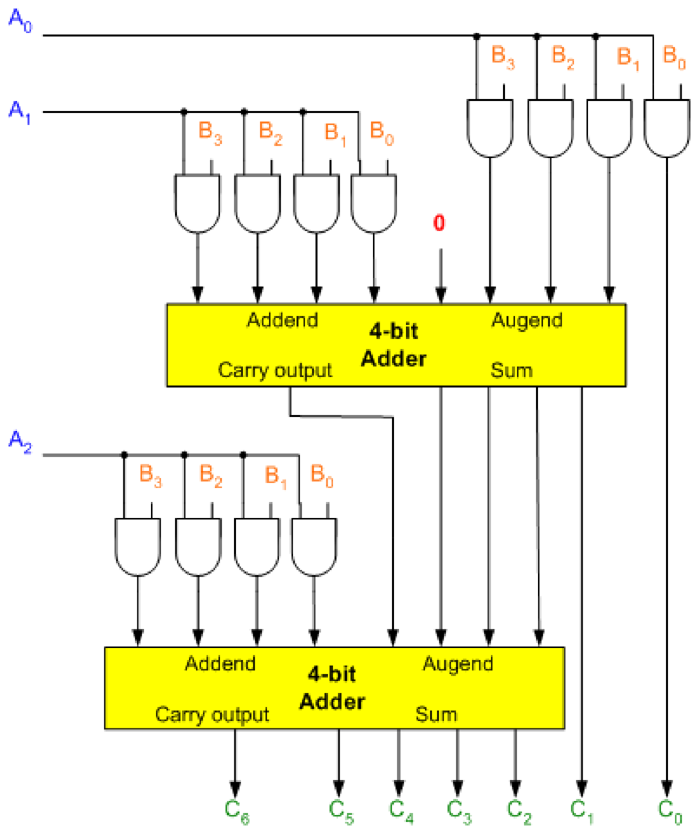
A binary multiplier with more bits can be constructed in a similar manner.

Example 2: Consider the example of multiplying two numbers, say A (3-bit number) and B (4-bit number).

Each bit of A (the multiplier) is ANDed with each bit of B (the multipcand) as shown in the Figure.

$$
\begin{array}{ccccccc}
 & & & B_3 & B_2 & B_1 & B_0 \\
 & & & & \times A_2 & A_1 & A_0 \\
\hline
 & & & A_0B_3 & A_0B_2 & A_0B_1 & A_0B_0 \\
 & & A_1B_3 & A_1B_2 & A_1B_1 & A_1B_0 & \\
 & A_2B_3 & A_2B_2 & A_2B_1 & A_2B_0 & & \\
\hline
C_6 & C_5 & C_4 & C_3 & C_2 & C_1 & C_0
\end{array}
$$

The binary output in each level of AND gates is added in parallel with the partial product of the previous level to form a new partial product. The last level produces the final product.



Since ***J*** = 3 and ***K*** = 4, **12** (***J*** x ***K***) **AND gates** and **two 4-bit** ((***J*** - ***1***) ***K*-bit**) **adders** are needed to produce a product of **seven** (***J*** + ***K***) bits. Its circuit is shown in the Figure.

Note that **0** is applied at the most significant bit of augend of first 4-bit adder because the least significant bit of the product does not have to go through an adder.