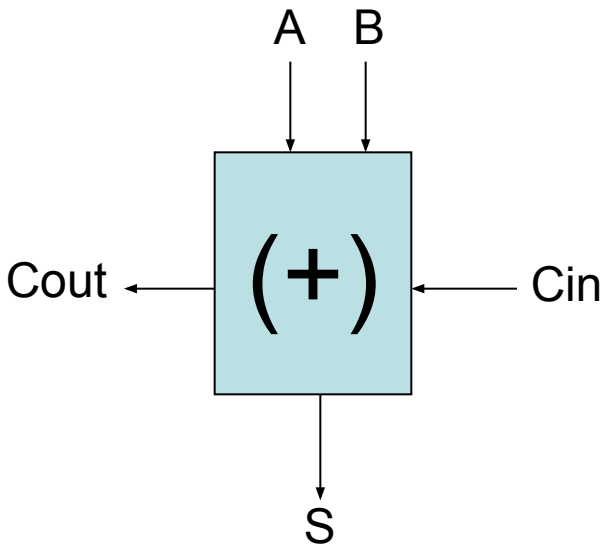
CARRY LOOK AHEAD ADDER
FLOATING POINT ARITHMETIC
IEEE 754 FORMAT

UNIT 2

FA

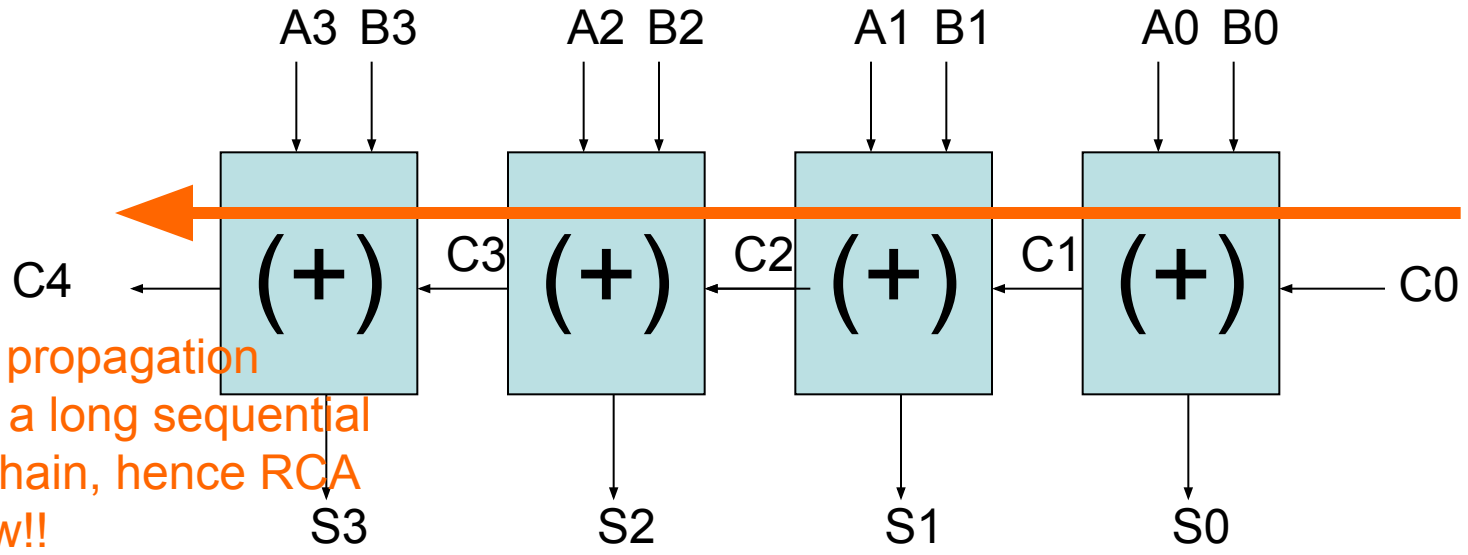
Inputs			Outputs	
A	B	C_{in}	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

A simple one-bit full adder



It takes A, B, and Cin as input and generates S and Cout in 2 gate delays (SOP)

4-bit RCA



Carry propagation forms a long sequential wait chain, hence RCA Is slow!!

- Work from lowest bit to highest bit sequentially.
- With A0, B0, and C0, the lowest bit adder generates S0 and C1 in 2 gate delay.
- With A1, B1, and C1 ready, the second bit adder generates S1 and C2 in 2 gate delay.
- Each bit adder has to wait for the lower bit adder to propagate the carry.

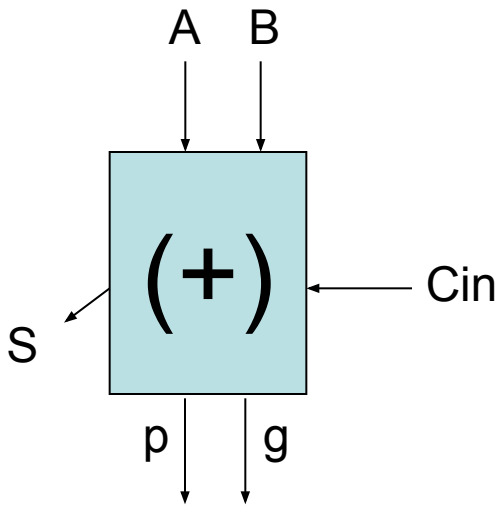
Observations

- The critical component each bit adder waits for is the carry input.
- Instead of generating and propagating carry bit-by-bit, can we generate all of them in parallel and break the sequential chain?
- This is exactly the idea of CLA (carry look-ahead adder).

Carry Look Ahead Logic

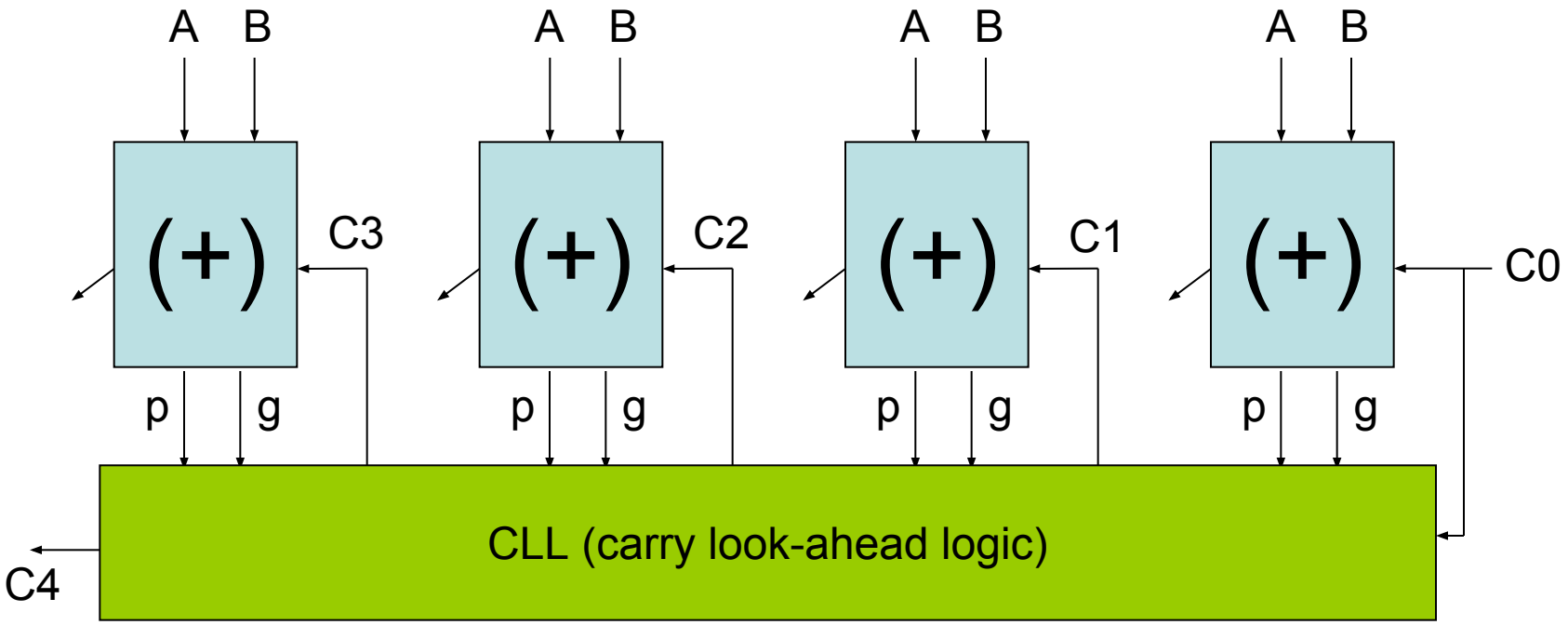
- Now even before the carry in (Cin) is available, based on the inputs (A,B) only, can we say anything about the carry out?
- Under what condition will the bit propagate an outgoing carry (Cout), if there is an incoming carry (Cin)?
- Under what condition will the bit generate an outgoing carry (Cout), regardless of whether there is an incoming carry (Cin)?

1-bit CLA adder



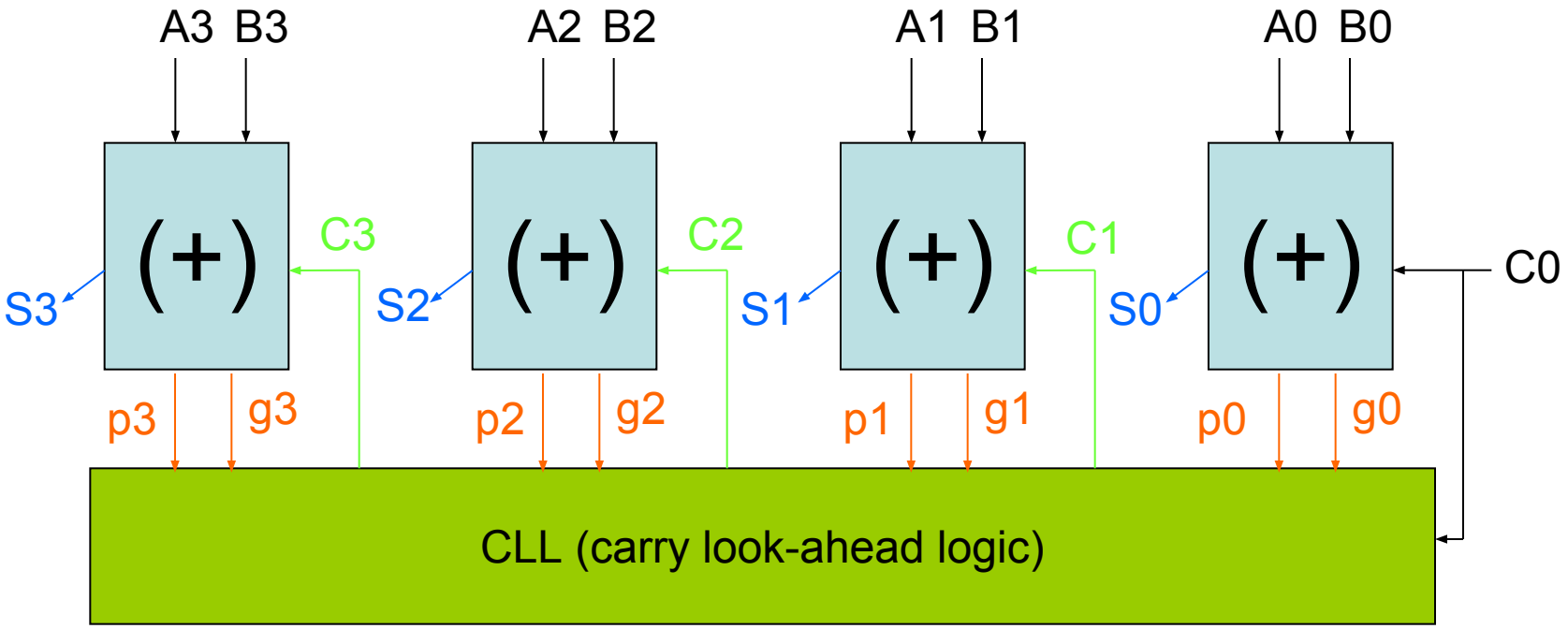
- Instead of Cout, an 1-bit CLA adder block takes A, B inputs and generates p,g
- p=propagator =>I will propagate the Cin to the next bit. p = A+B
  (If either A or B is 1, Cin=1 causes Cout=1)
- g=generator =>I will generate a Cout independent of what Cin is. g = AB
  (If both A and B are 1, Cout=1 for sure)
- p,g are generated in 1 gate delay after we have A,B. Note that Cin is not needed to generate p,g.
- S is generated in 2 gate delay after we get Cin (SOP).

4-bit CLA



- The CLL takes p,g from all 4 bits and C0 as input to generate all Cs in 2 gate delay.
- C1=g0+p0C0,
- C2=g1+p1g0+p1p0C0,
- C3=g2+p2g1+p2p1g0+p2p1p0c0,
- C4=g3+p3g2+p3p2g1+p3p2p1g0+p3p2p1p0c0 (Note: this C4 is too complicated to generate in 2-level SOP representation)

4-bit CLA



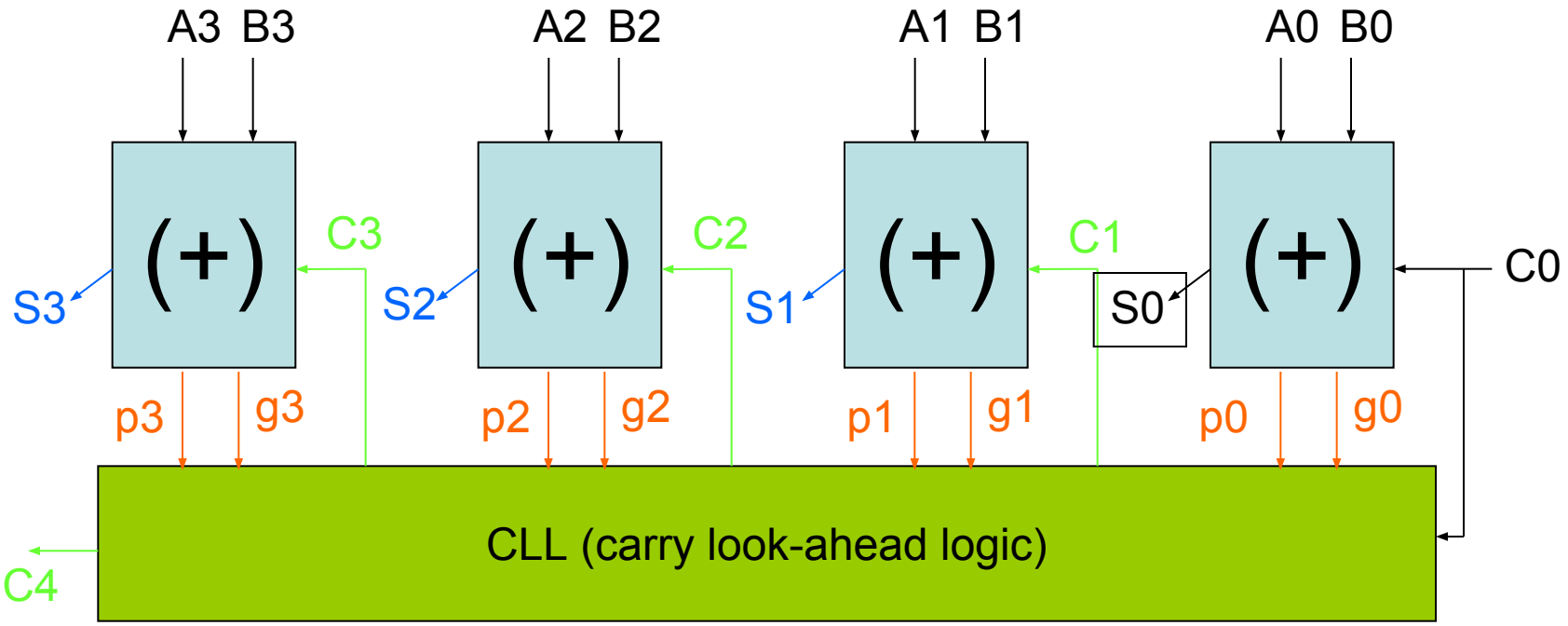
- Given A,B's, all p,g's are generated in 1 gate delay in parallel.
- Given all p,g's, all C's are generated in 2 gate delay in parallel.
- Given all C's, all S's are generated in 2 gate delay in parallel.

- Key virtue of CLA: sequential operation in RCA is broken into parallel operation!!

Observation

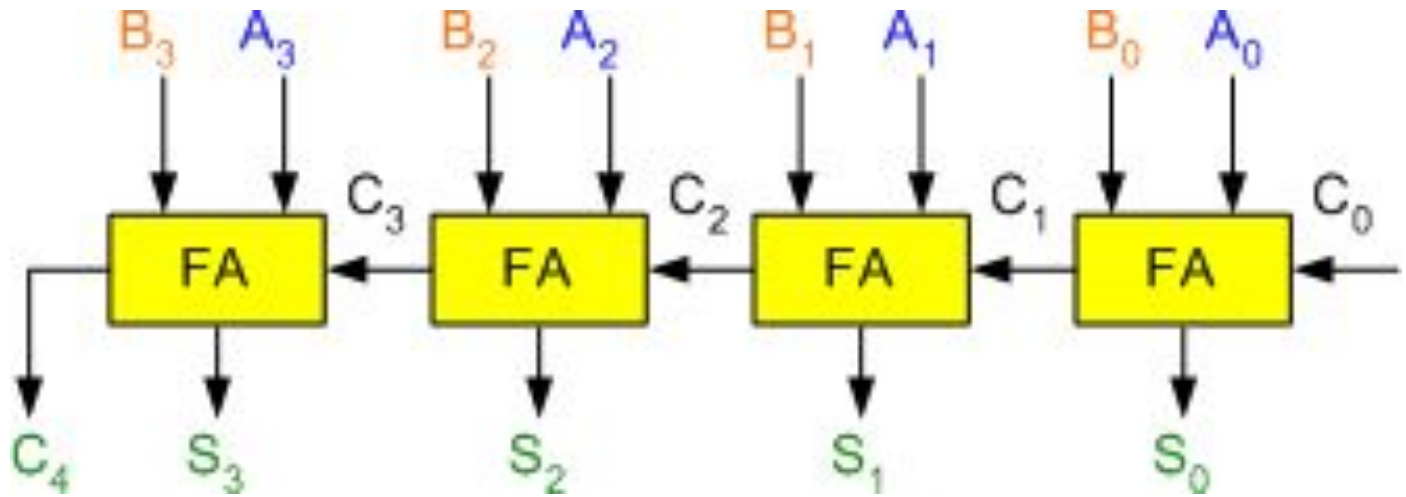
- The CLL block cannot be made too big (at most 4 bits) because if the equations for C's are too long it cannot be evaluated in 2 gate delay.
- So how about long operands, say 16 bits?
- We add another layer of CLL and make a multi-level CLA.

A bit more details



- Do all these 4 S's (S3, S2, S1, S0) come together?
  Actually no! Since C0 is available from the beginning, S0 can be calculated in 2 gate delays (using original SOP expression for S bit in a single bit adder) (before S3,S2,S1)

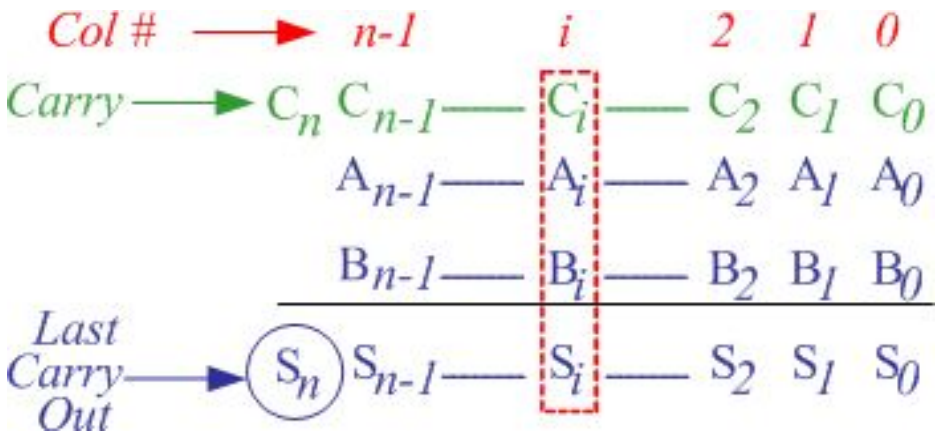
- In ***ripple carry adders***, the carry propagation time is the major speed limiting factor as seen in the previous lesson.



- 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:

when both bits A_i and B_i are 1, or

when one of the two bits is 1 and the carry-in (carry of the previous stage) is 1.



- ***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.

•

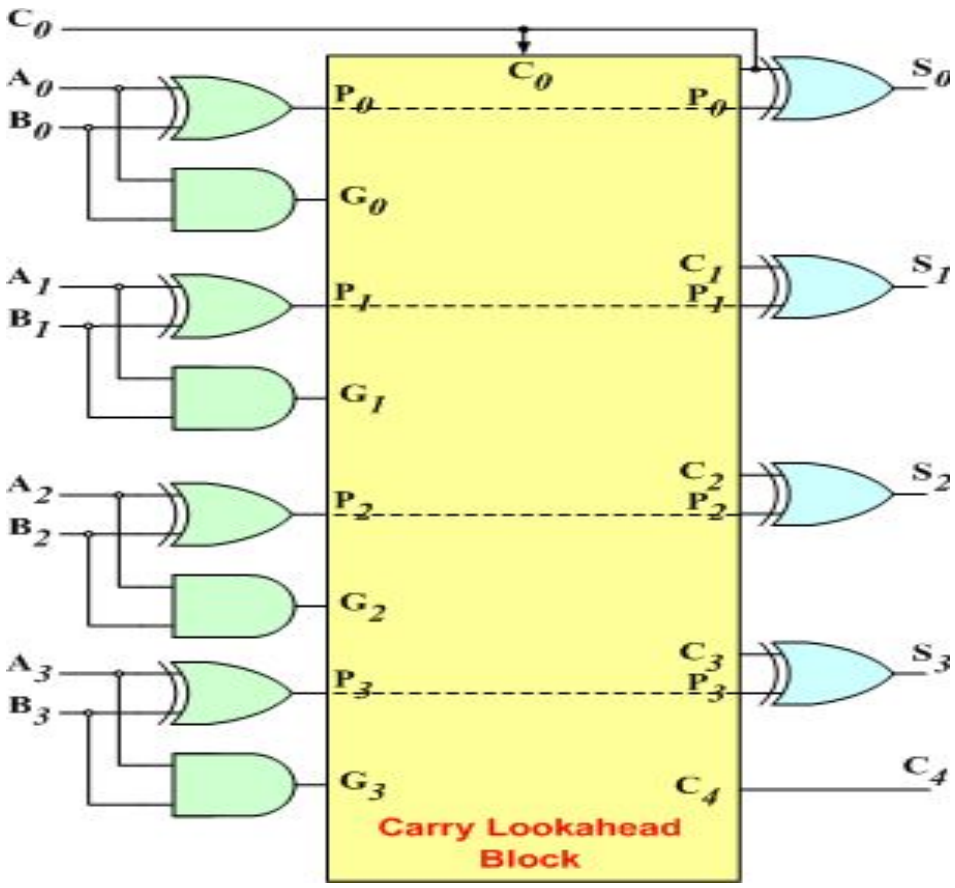
•Computed values of ***all*** the G_i's are valid one AND-gate delay after the operands A and B are made valid.

•

- The Boolean expression of the carry outputs of various stages can be written as follows:
- $C_1 = G_0 + P_0C_0$
- $C_2 = G_1 + P_1C_1 = G_1 + P_1 (G_0 + P_0C_0)$
- $= G_1 + P_1G_0 + P_1P_0C_0$
- $C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$ $C_4 = G_3 + P_3C_3$
- $= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$

- In general, the $i^{th.}$ carry output is expressed in the form $\mathbf{C_i} = F_i$ (P's, G's , $\mathbf{C_0}$).
- In other words, each carry signal is expressed as a direct SOP function of $\mathbf{C_0}$ 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., 2 $\tilde{}$

Carry look ahead adder



- **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 1□.
-
- **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 3□.
-

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

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

•

•For a 4-bit adder (*n* = 4), the Ripple Carry adder delay is 9□.

•

•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.

•

FLOATING POINT ARITHMETIC

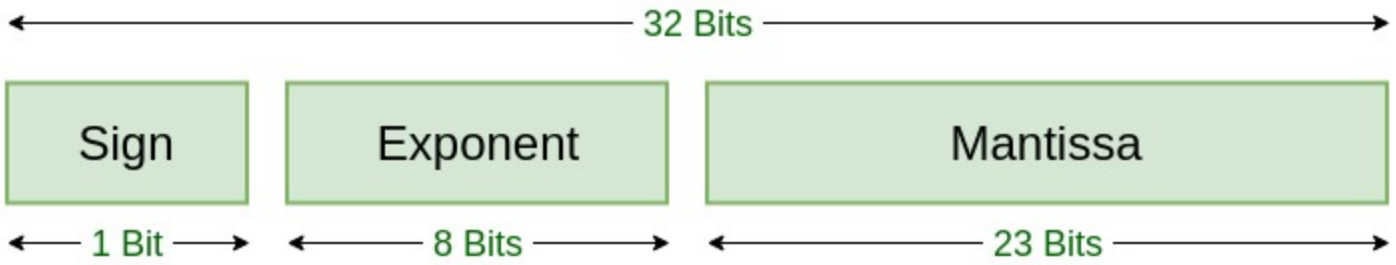
IEEE Standard for Floating point Number

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point computation which was established in 1985 by the **Institute of Electrical and Electronics Engineers (IEEE)**. The standard addressed many problems found in the diverse floating point implementations that made them difficult to use reliably and reduced their portability. IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macs, and most Unix platforms.

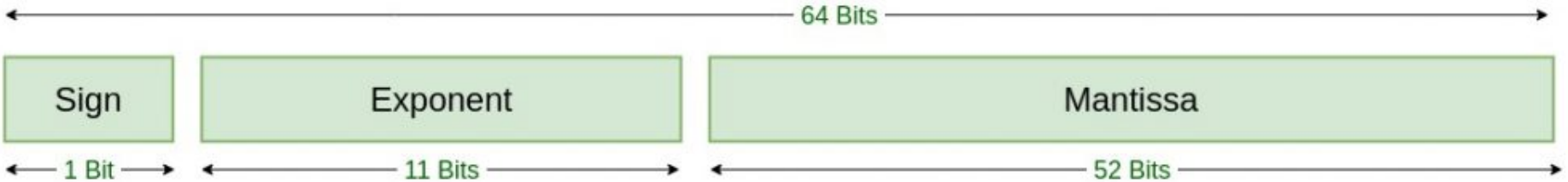
- There are several ways to represent floating point number but IEEE 754 is the most efficient in most cases. IEEE 754 has 3 basic components:

1. **The Sign of Mantissa –**
   This is as simple as the name. 0 represents a positive number while 1 represents a negative number.
2. **The Biased exponent –**
   The exponent field needs to represent both positive and negative exponents. A bias is added to the actual exponent in order to get the stored exponent.
3. **The Normalised Mantissa –**
   The mantissa is part of a number in scientific notation or a floating-point number, consisting of its significant digits. Here we have only 2 digits, i.e. O and 1. So a normalised mantissa is one with only one 1 to the left of the decimal.

- **IEEE 754 numbers are divided into two based on the above three components: single precision and double precision.**
-



Single Precision
IEEE 754 Floating-Point Standard



Double Precision
IEEE 754 Floating-Point Standard

TYPES	SIGN	BIASED EXPONENT	NORMALISED MANTISA	BIAS
Single precision	1(31st bit)	8(30-23)	23(22-0)	127
Double precision	1(63rd bit)	11(62-52)	52(51-0)	1023

example

- Ques : 1 00000100 00000000000……111
- Represent 32 bit single precision floating point no
- Sol : number is –ve
- Exponent 4
- Ans – 15 *10 to the power -4
- =-.0015

example

Ques 32 bit single precesion

0 00000011 00000000000000000001110

Ques - .00012

BINARY REPRESENTATION OF FLOATING POINT NUMBERS

Converting decimal fractions into binary representation.

Consider a decimal fraction of the form: 0.d1d2...dn

We want to convert this to a binary fraction of the form:

0.b1b2...bn (using binary digits instead of decimal digits)

COMPUTER REPRESENTATION OF FLOATING POINT NUMBERS

In the CPU, a 32-bit floating point number is represented using IEEE standard format as follows:

S | EXPONENT | MANTISSA

where S is one bit, the EXPONENT is 8 bits, and the MANTISSA is 23 bits.

- The ***mantissa*** represents the leading significant bits in the number.

- The ***exponent*** is used to adjust the position of the binary point (as opposed to a "decimal" point)

The mantissa is said to be **normalized** when it is expressed as a value between 1 and 2. I.e., the mantissa would be in the form 1.xxxx.

The leading integer of the binary representation is not stored. Since it is always a 1, it can be easily restored.

The "S" bit is used as a sign bit and indicates whether the value represented is positive or negative

(0 for positive, 1 for negative).

If a number is smaller than 1, normalizing the mantissa will produce a negative exponent.

But 127 is added to all exponents in the floating point representation, allowing all exponents to be represented by a positive number.