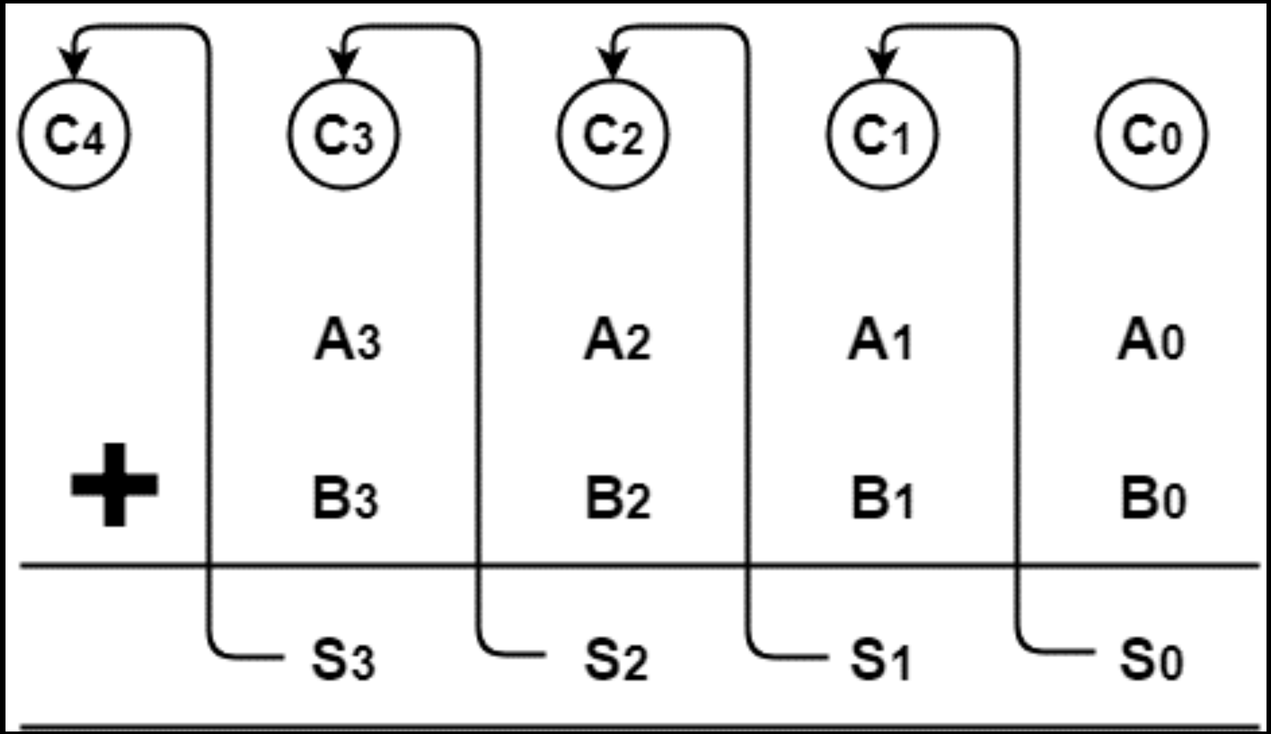
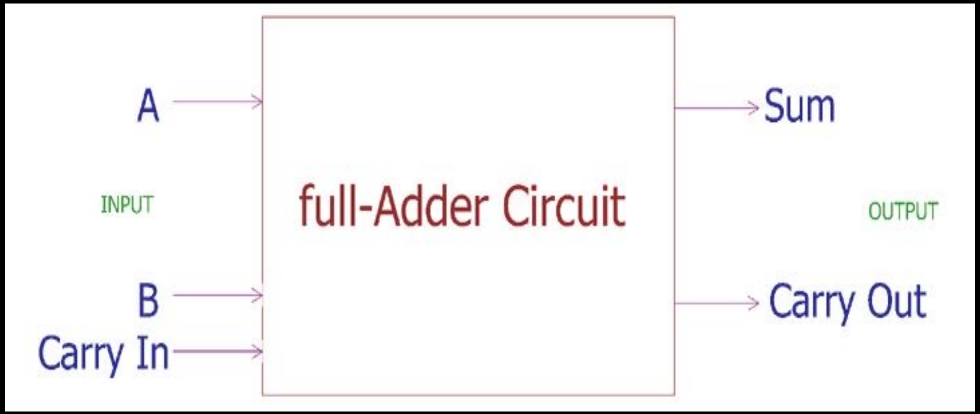
Full adder

1. A full adder is a combinational logic circuit that performs the arithmetic sum of three input bits.
2. Where A_n, B_n are the nth order bits of the number A and B respectively and C_n is the carry generated from the addition of (n-1)th order bits.
3. It consists of three input bits, denoted by A (First operand), B (Second operand), C_{in} (Represents carry from the previous lower significant position).



- Two output bits are same as of half adder, which is Sum and Carry$_{out}$.
- When the augend and addend number contain more significant digits, the carry obtained from the addition of two bits is added to the next higher order pair of significant bits.

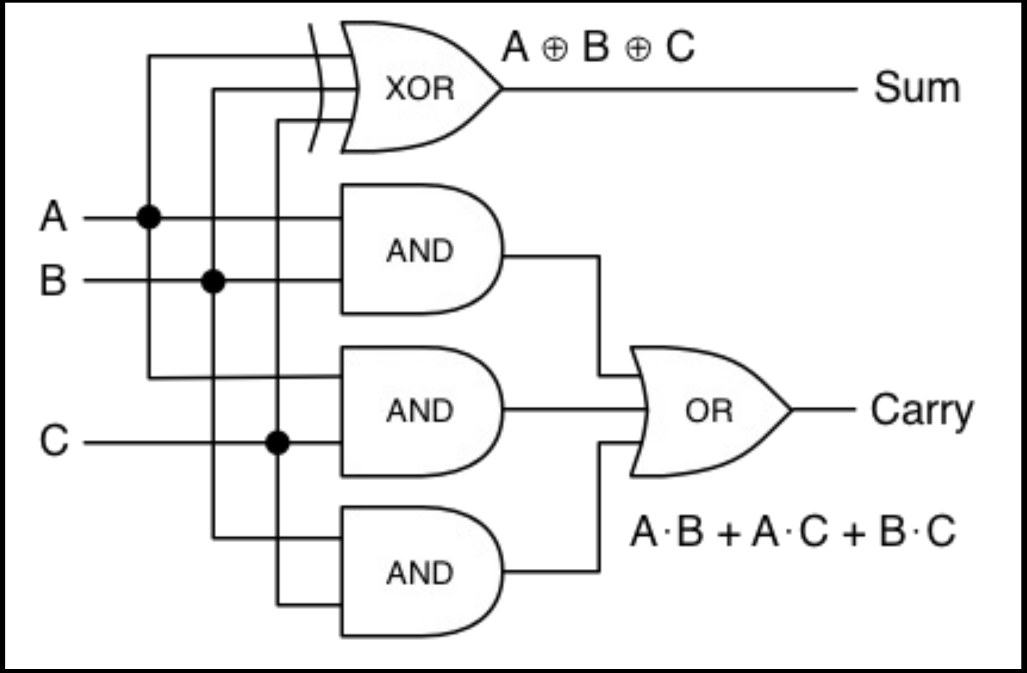


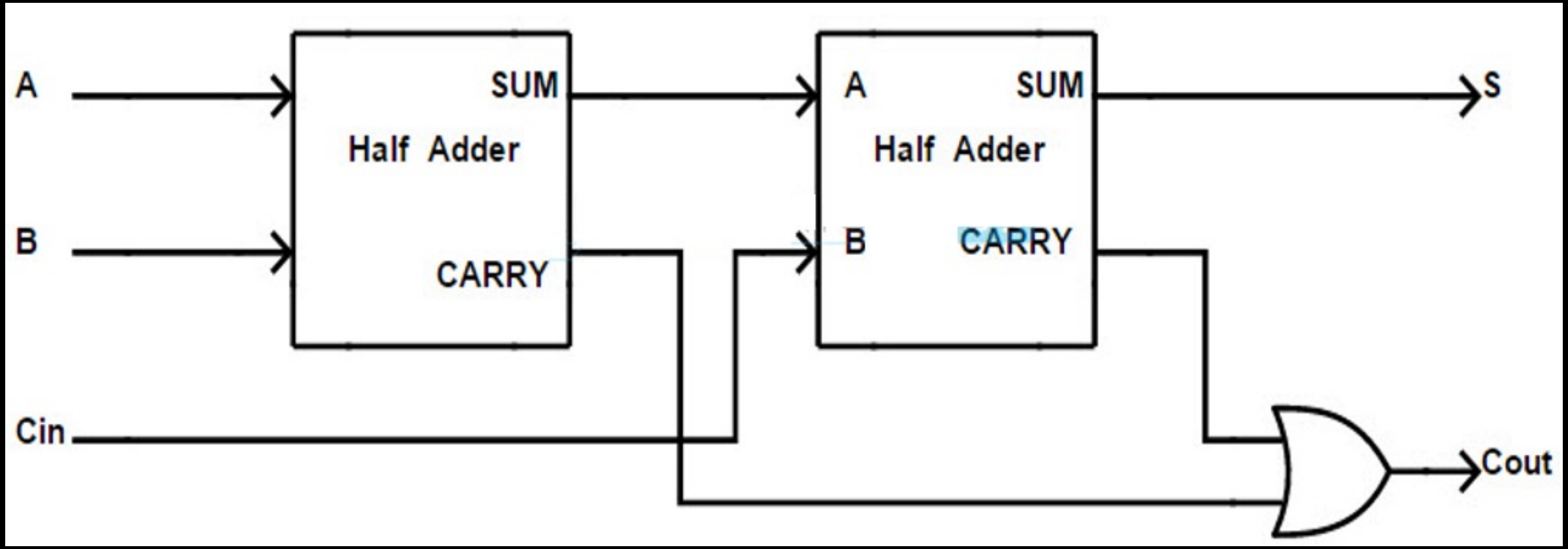
INPUTS			OUTPUTS	
A	B	C $_{in}$	C $_{out}$	Sum
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

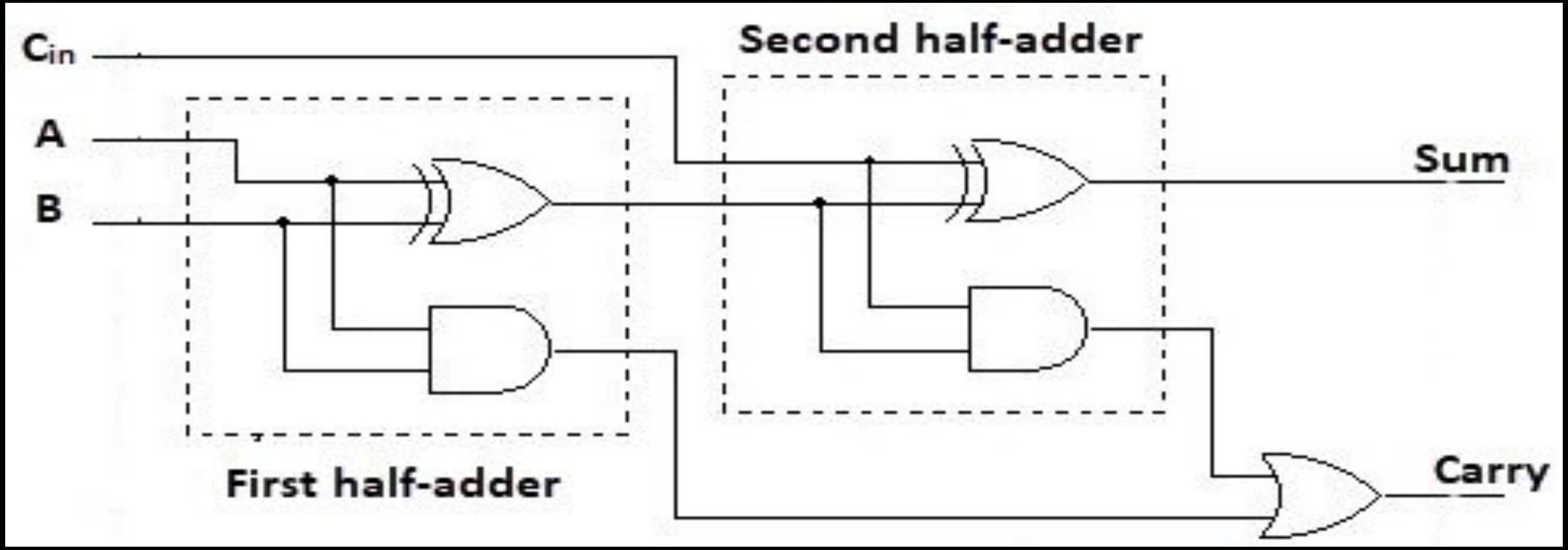
	ab	a’b’	a’b	ab	ab’
c_{in}		00	01	11	10
c_{in}'	0	0	2	6	4
c_{in}	1	1	3	7	5

	ab	a’b’	a’b	ab	ab’
c_{in}		00	01	11	10
c_{in}'	0	0	2	6	4
c_{in}	1	1	3	7	5

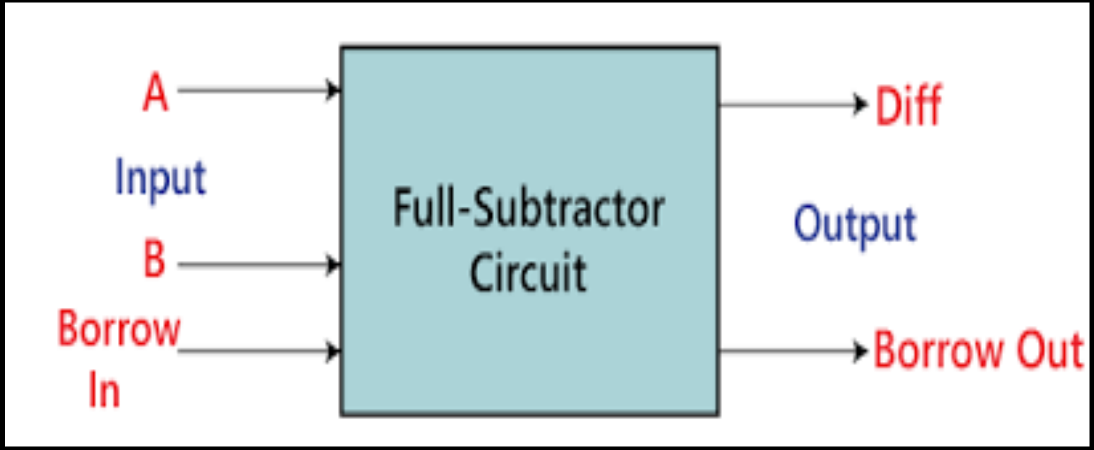
INPUTS			OUTPUTS	
A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

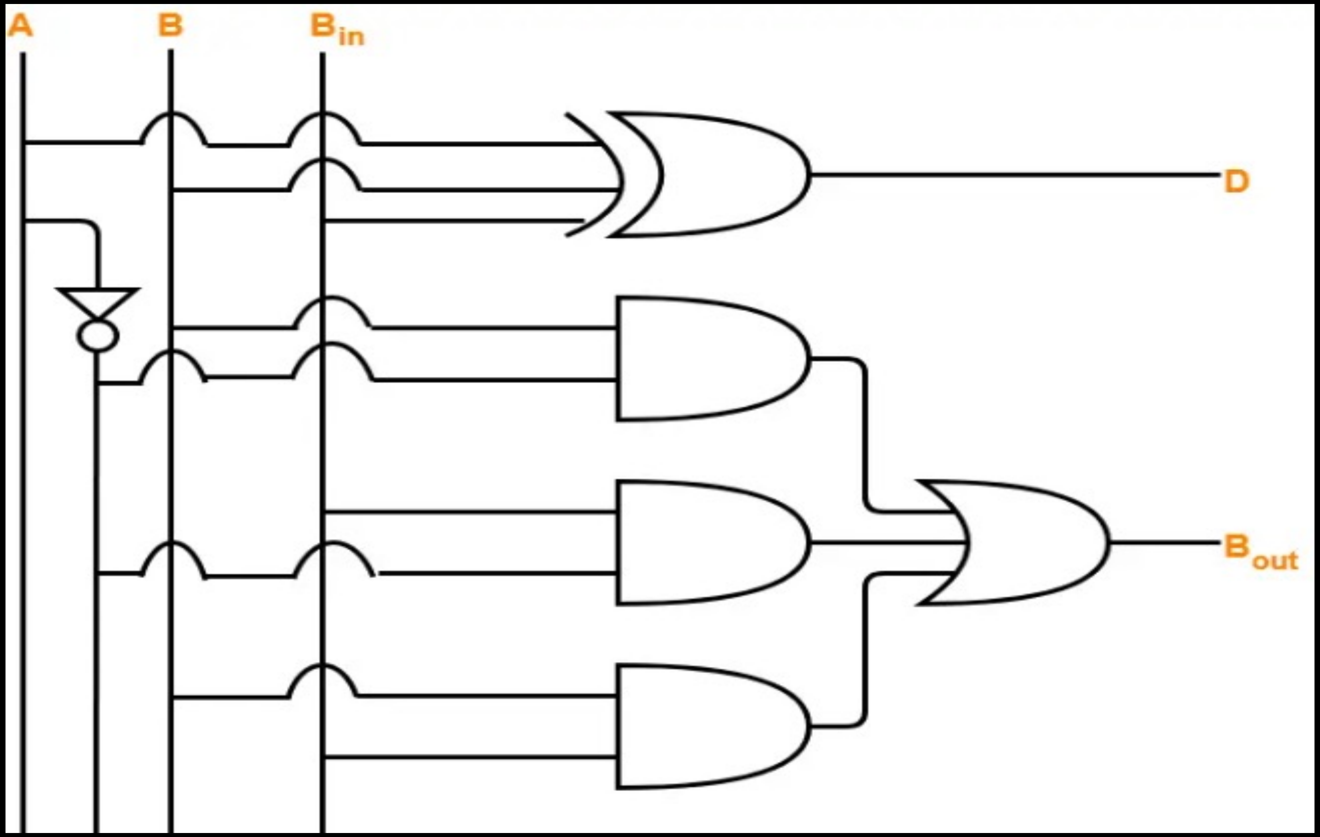


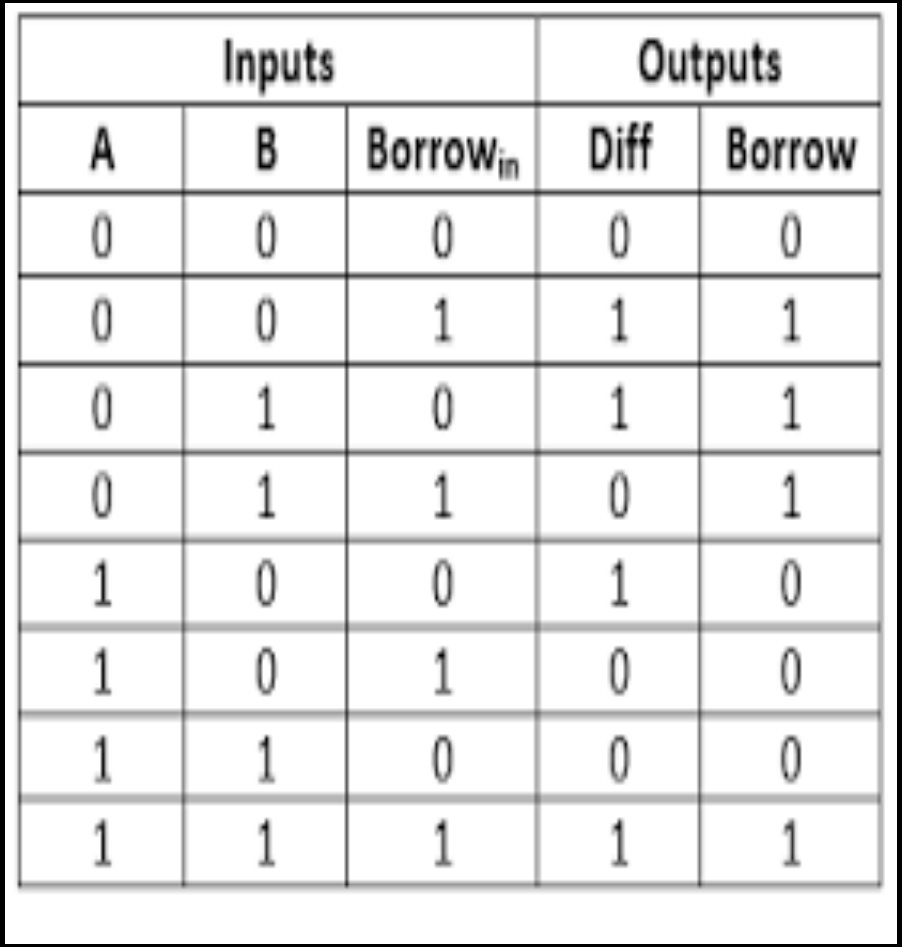

A
B
Cin
Half Adder
SUM
CARRY
A
B
Half Adder
SUM
CARRY
S
Cout


Cin
A
B
Second half-adder
First half-adder
Sum
Carry

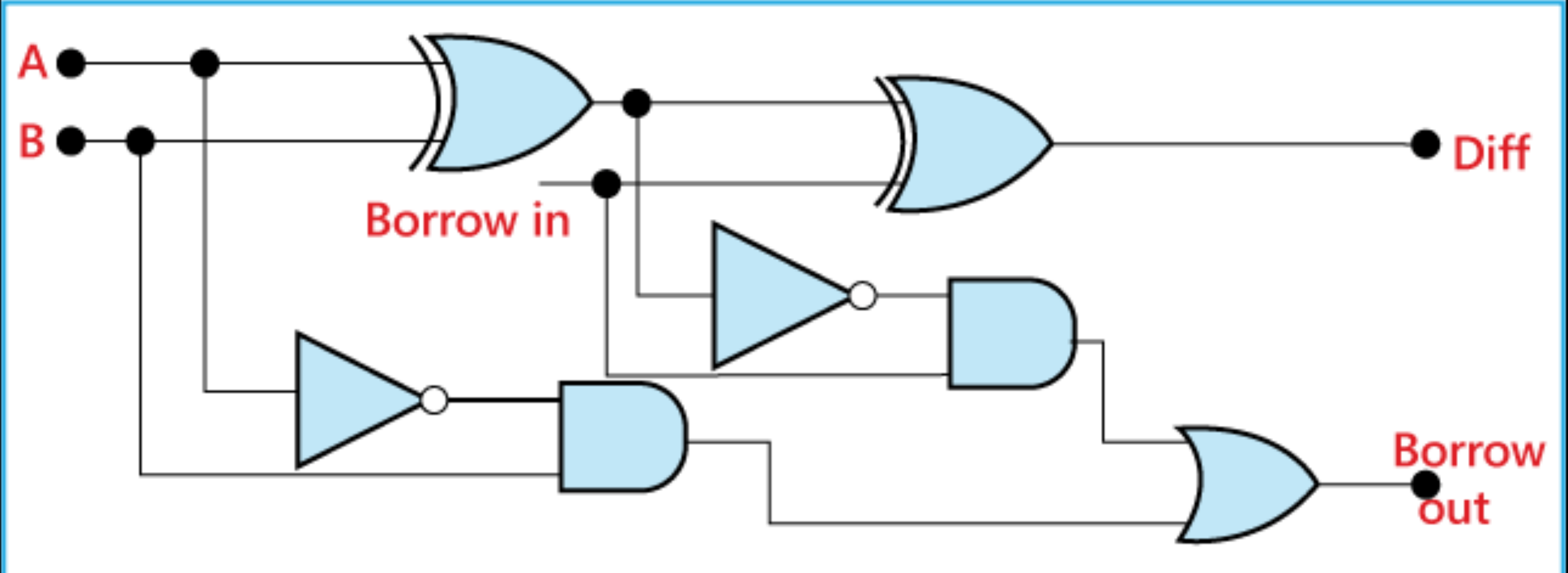
Full subtractor







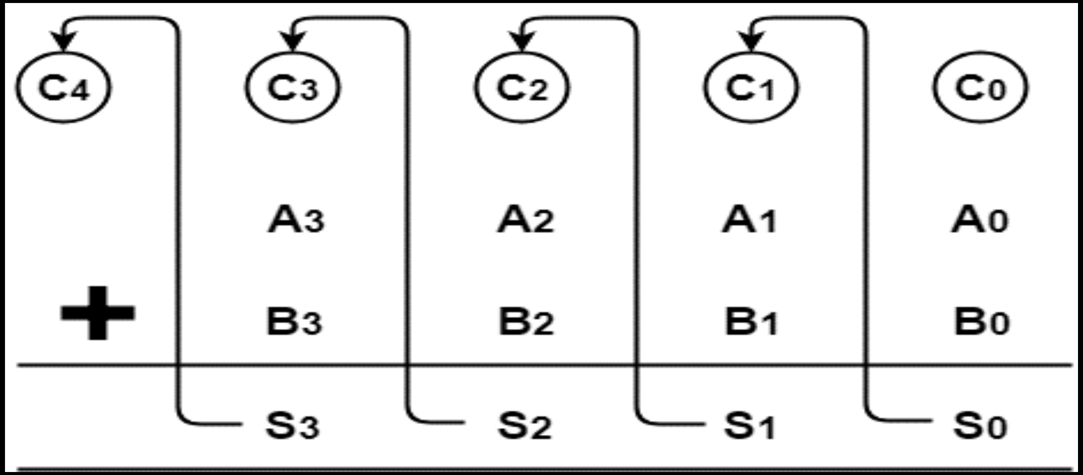
Inputs			Outputs	
A	B	Borrow_{in}	Diff	Borrow
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

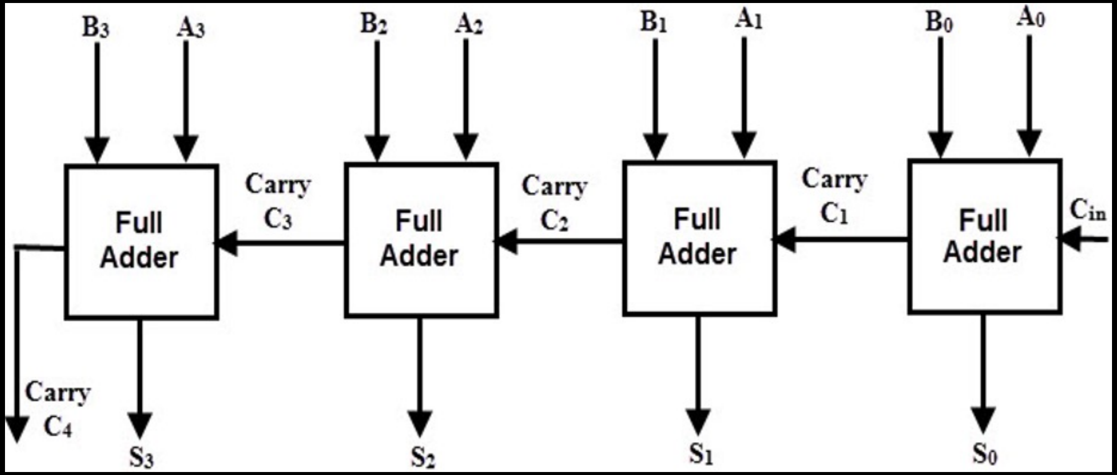


Full-Subtractor Circuit

Four-bit parallel binary adder / Ripple adder

- As we know that full adder is capable of adding two 1 bit number and 1 previous carry, but in order to add binary numbers with more than one bits, additional full adders must be employed. For e.g. a four bit binary adder can be constructed using four full adders.
- Theses four full adders are connected in cascade, carry output of each adder is connected to the carry input of the next higher-order adder. So a n-bit parallel adder is constructed using 'n' number of full adders.





- There are some scope of improvement in parallel binary adder / Ripple adder is it is very slow

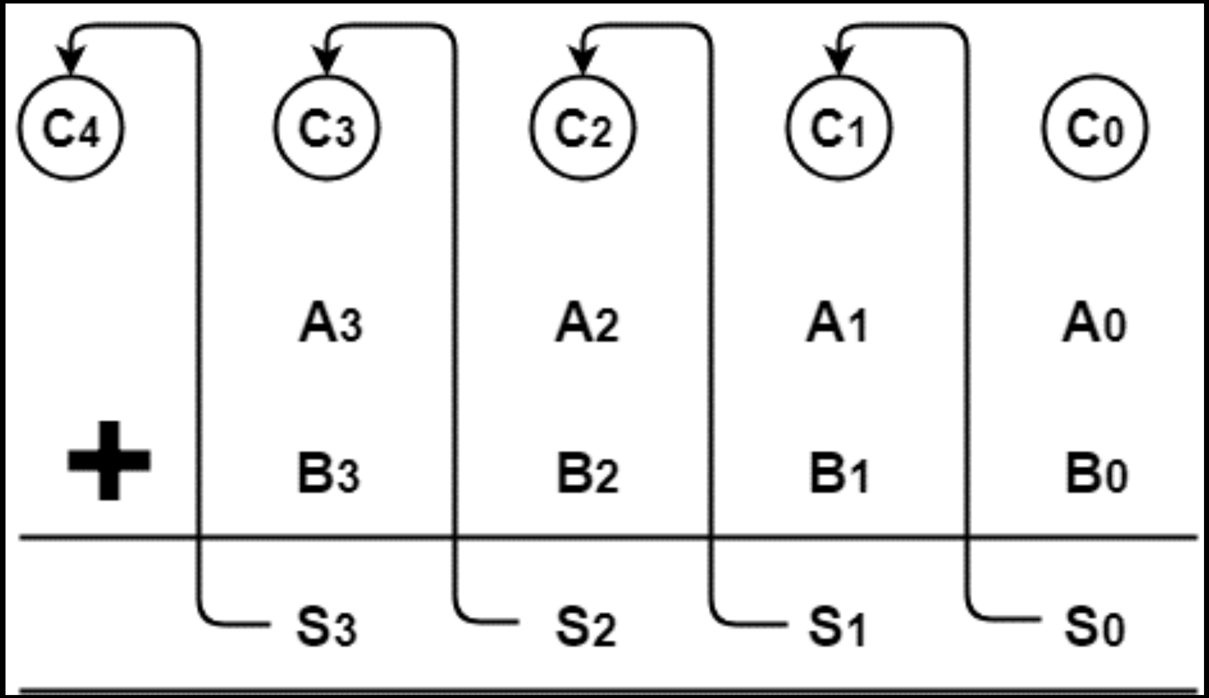
Carry propagation delay

Look ahead Carry Generator

Look ahead carry adder

- The carry propagation time is an important attribute of the adder because it limits the speed with which two numbers are added. The solution to delay is to increase the complexity of the equipment in such a way that the carry delay time is reduced.

- To solve this problem most widely used technique employs the principle of 'look ahead carry'. This method utilizes logic gates to look at the lower order bits of the augend and addend to see if a higher order carry is to be generated. It uses two functions carry generate G_i and carry propagate P_i



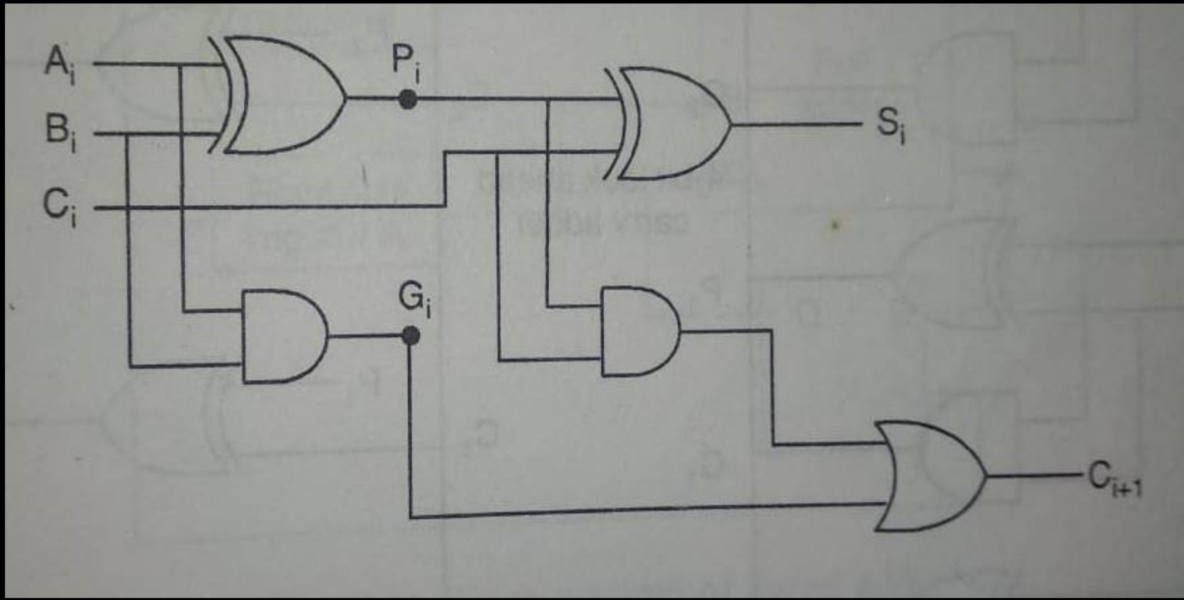
$A(A_3\ A_2\ A_1\ A_0)$

$B(B_3\ B_2\ B_1\ B_0)$

- G_i is called a ***carry generate***, and it produces a carry of 1 when both A_i and B_i are 1, regardless of the input carry C_i.

- P_i is called a ***carry propagate***, because it determines whether a carry into stage i will propagate into stage i + 1.

- We now write the Boolean functions for the carry outputs of each stage and substitute the value of each C_i from the previous equations:

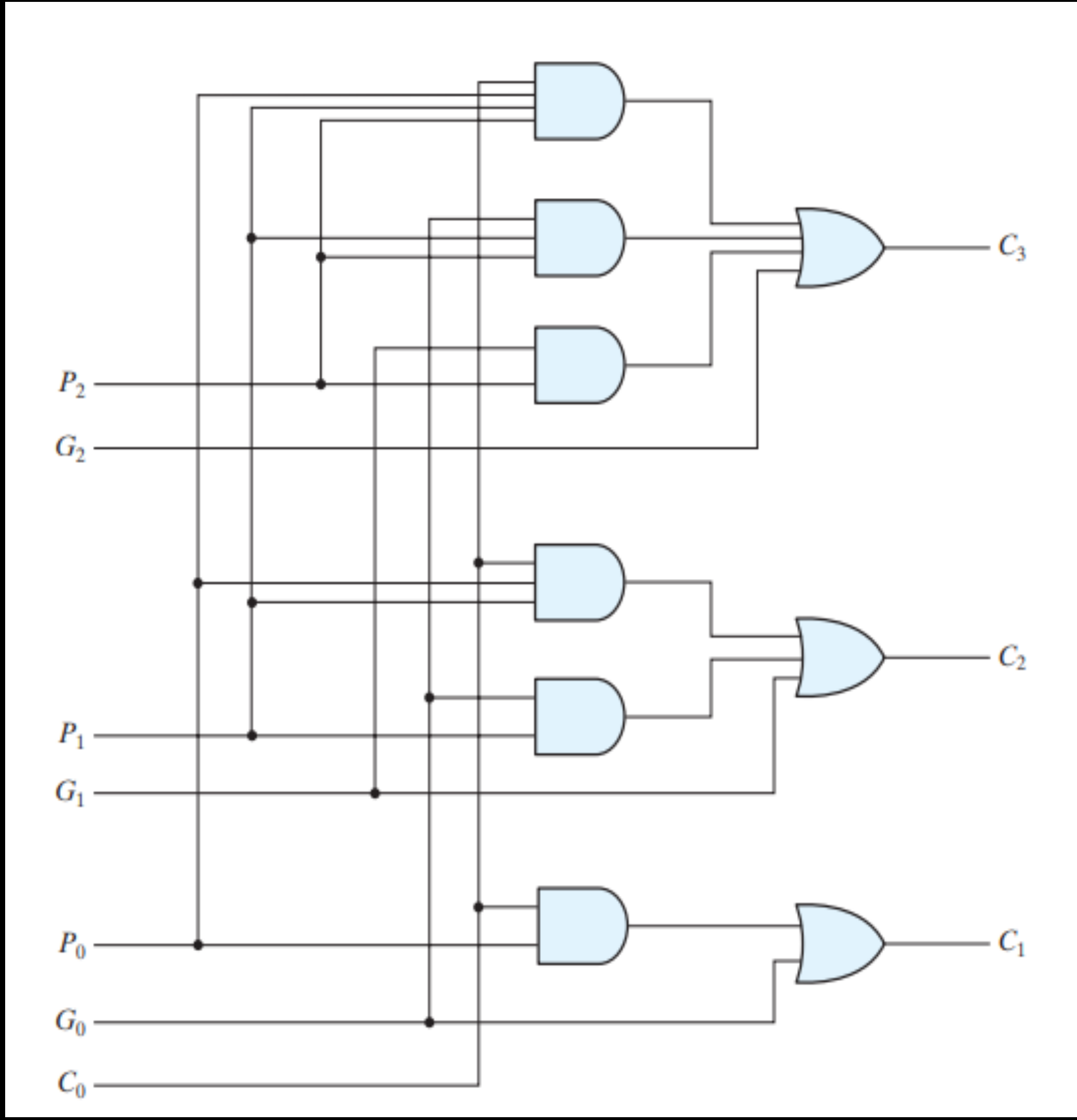


- $P_i = A_i \oplus B_i$
  $G_i = A_i . B_i$
  $S_i = P_i \oplus C_i$
  $C_{i+1} = G_i + P_i C_i$

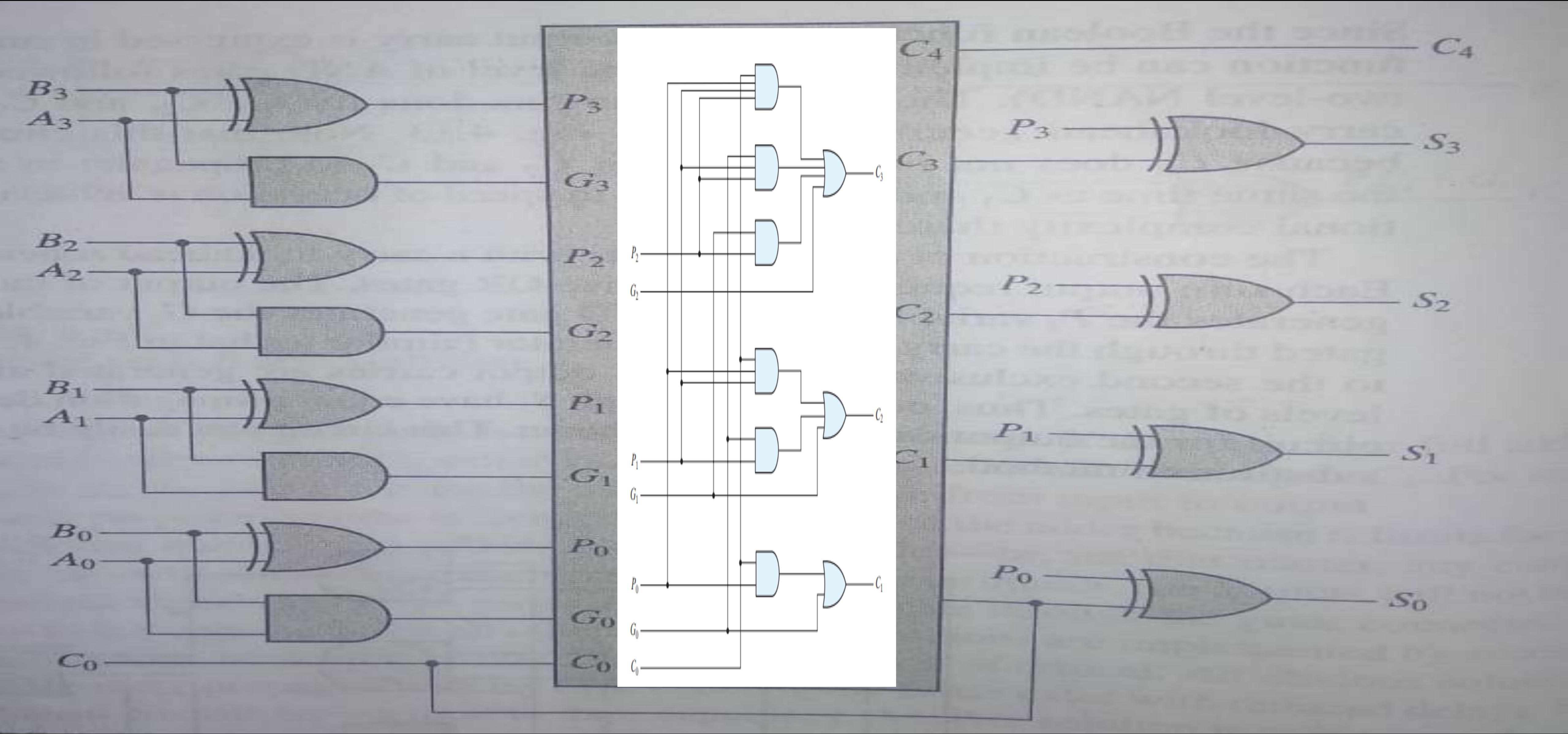
- $C_0 = 0$
- $C_1 = G_0 + P_0 C_0$
- $C_2 = G_1 + P_1G_0 + P_1P_0 C_0$
- $C_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1 P_0 C_0$
- $C_4 = G_3 + G_2.P_3 + G_1.P_2.P_3 + G_0.P_1.P_2.P_3 + C_0.P_0.P_1.P_2.P_3$

- Since the Boolean function for each output carry is expressed in sum-of-products form. Each function can be implemented with one level of AND gates followed by an OR gate (or by a two-level NAND).

- $C_0 = 0$
- $C_1 = G_0 + P_0\ C_0$
- $C_2 = G_1 + P_1G_0 + P_1P_0\ C_0$
- $C_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1\ P_0\ C_0$
- $C_4 = G_3 + G_2.P_3 + G_1.P_2.P_3 + G_0.P_1.P_2.P_3 + C_0.P_0.P_1.P_2.P_3$

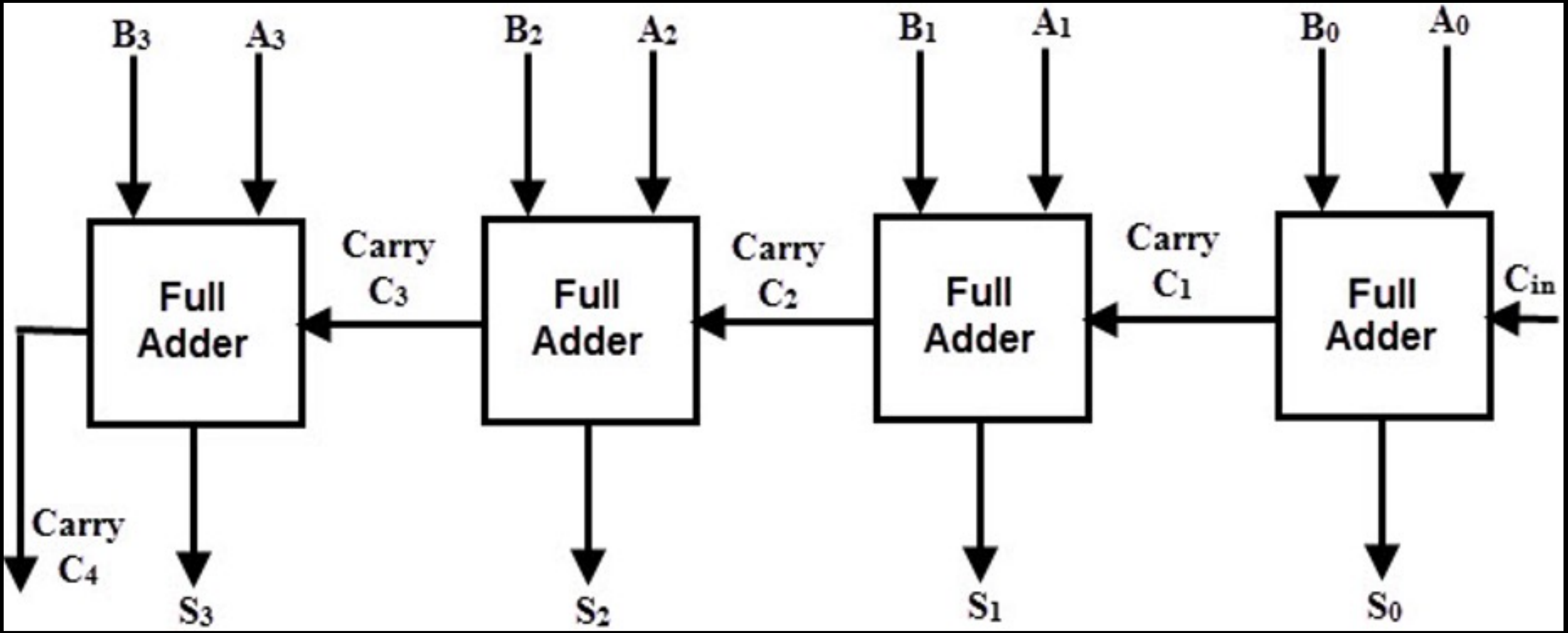


- All output carries are generated after a delay through two levels of gates. Thus, outputs S_1 through S_3 have equal propagation delay times.

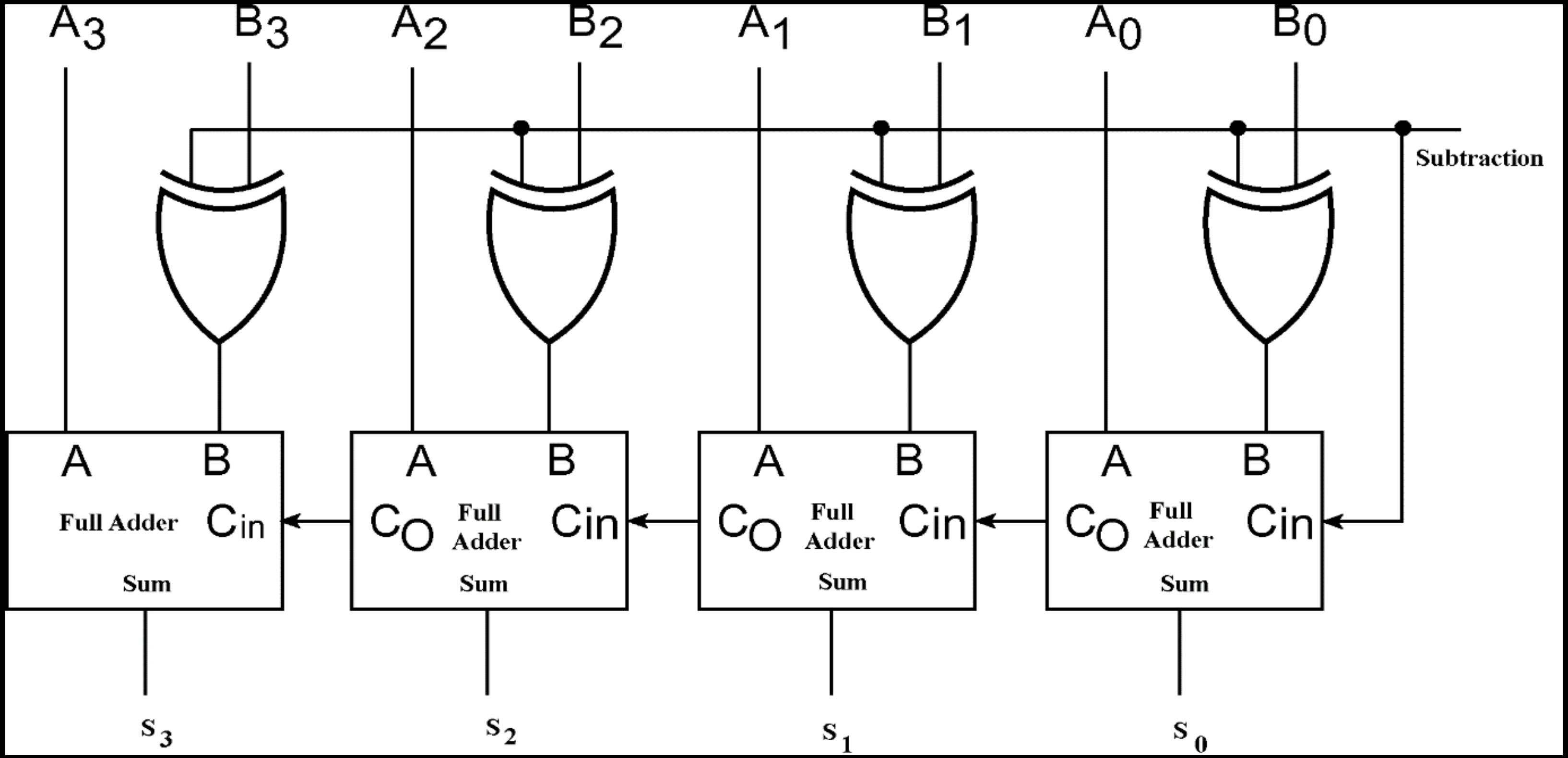


Four-bit ripple adder/subtractor

- The subtraction A - B can be done by taking the 2's complement of B and adding it to A.
- A + (-B)



- The circuit for subtracting A - B consists of an adder with inverters placed between each data input B and the corresponding input of the full adder. The mode input M controls the operation. When M = 0, the circuit is an adder, and when M = 1, the circuit becomes a subtractor.
- When M = 0, we have $B \oplus 0 = B$. The full adders receive the value of B, the input carry is 0, and the circuit performs A plus B.
- When M = 1, we have $B \oplus 1 = B'$ and $C_0 = 1$. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the 2's complement of B.



Booth’s algorithm

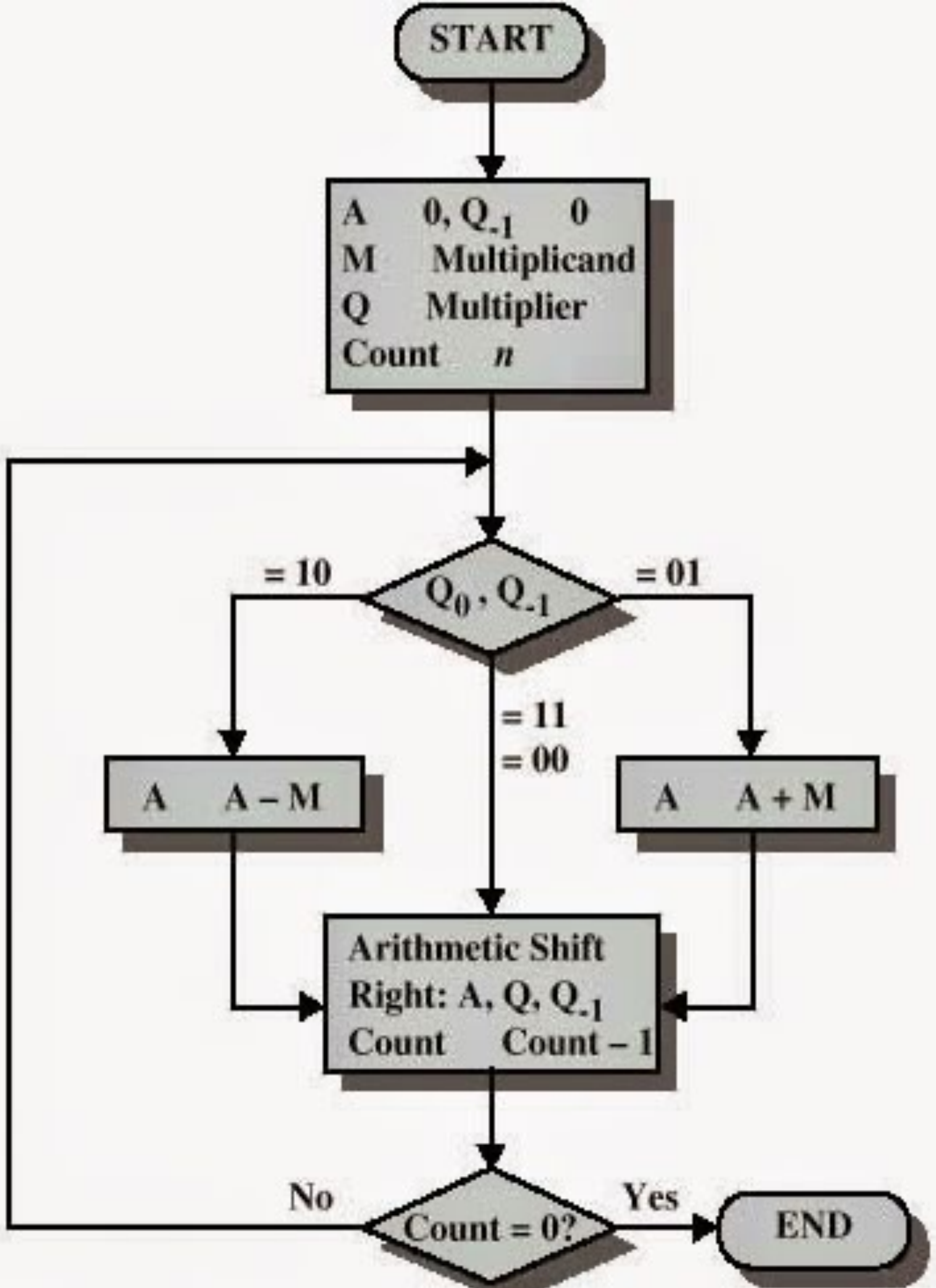
- **Andrew Donald Booth** (11 February 1918 – 29 November 2009) was a British electrical engineer, physicist and computer scientist, who was an early developer of the magnetic drum memory for computers. He is known for Booth's multiplication algorithm.

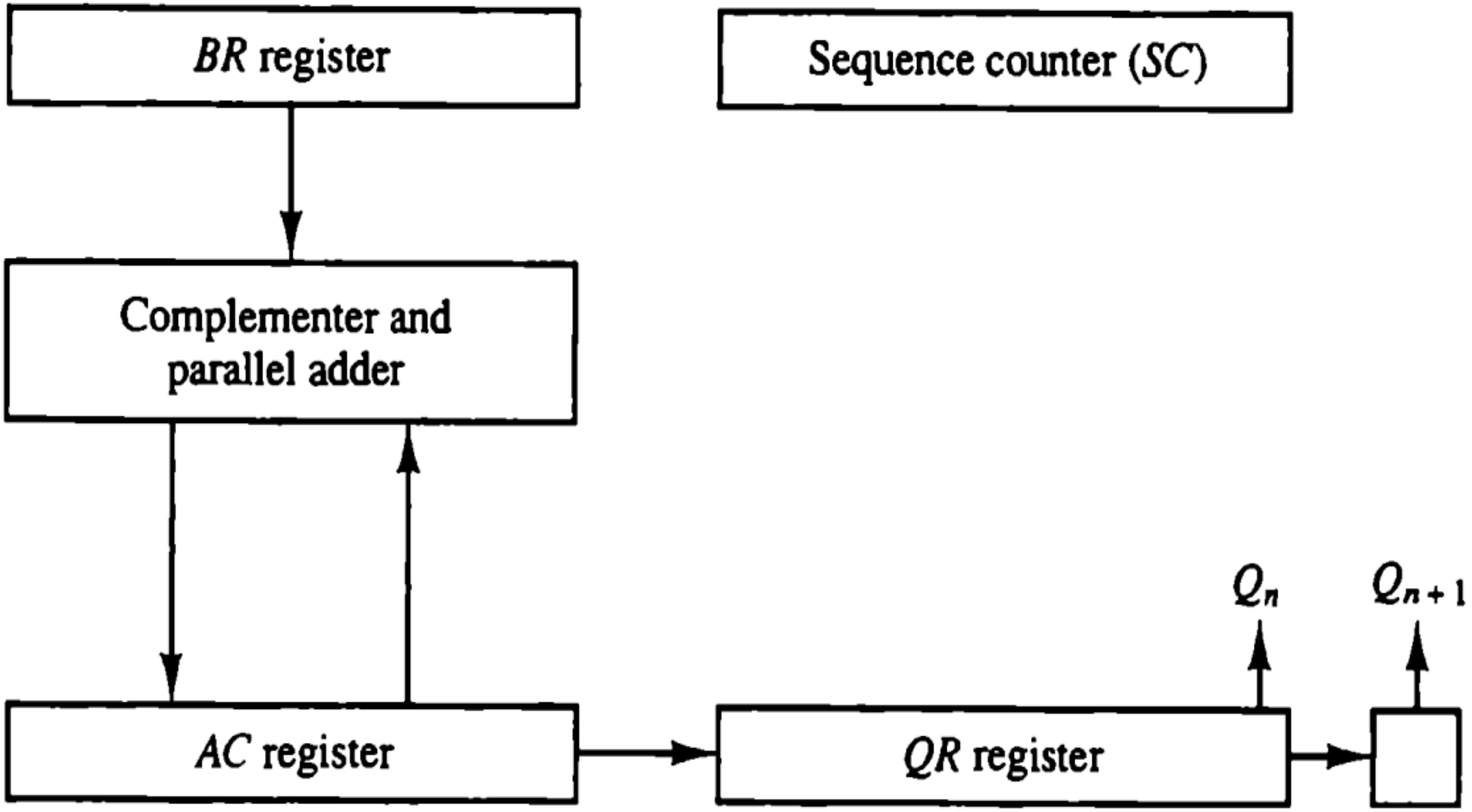
- Booth algorithm optimizes binary multiplication by reducing the number of additions and subtractions based on the multiplier bits.

- The process involves examining multiplier bits, then either adding, subtracting, or leaving the multiplicand unchanged before shifting the partial product.

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial values	
1001	0011	0	0111	A ← A − M	First cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second cycle
0101	0100	1	0111	A ← A + M	Third cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth cycle

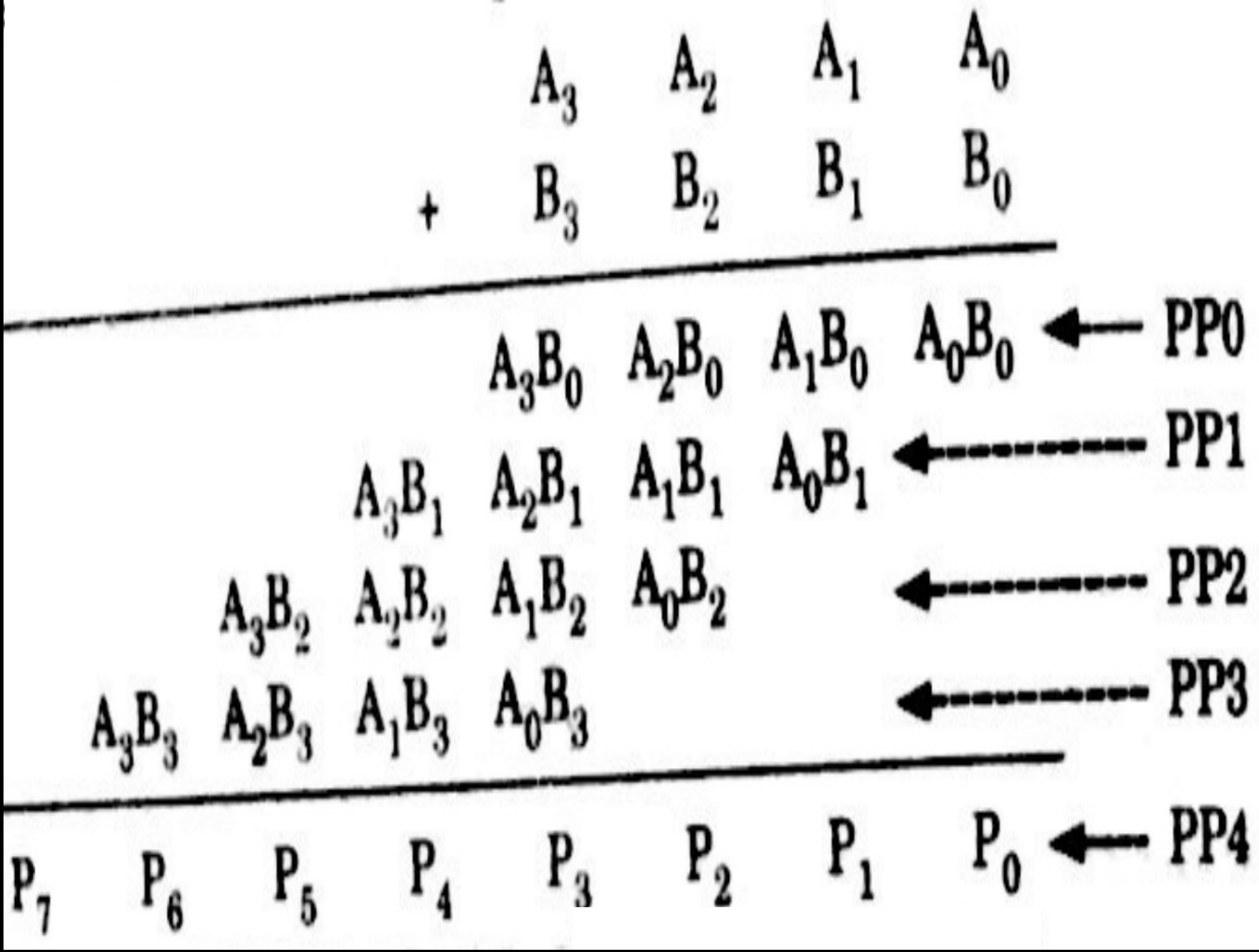
Example of Booth's Algorithm (7 × 3)



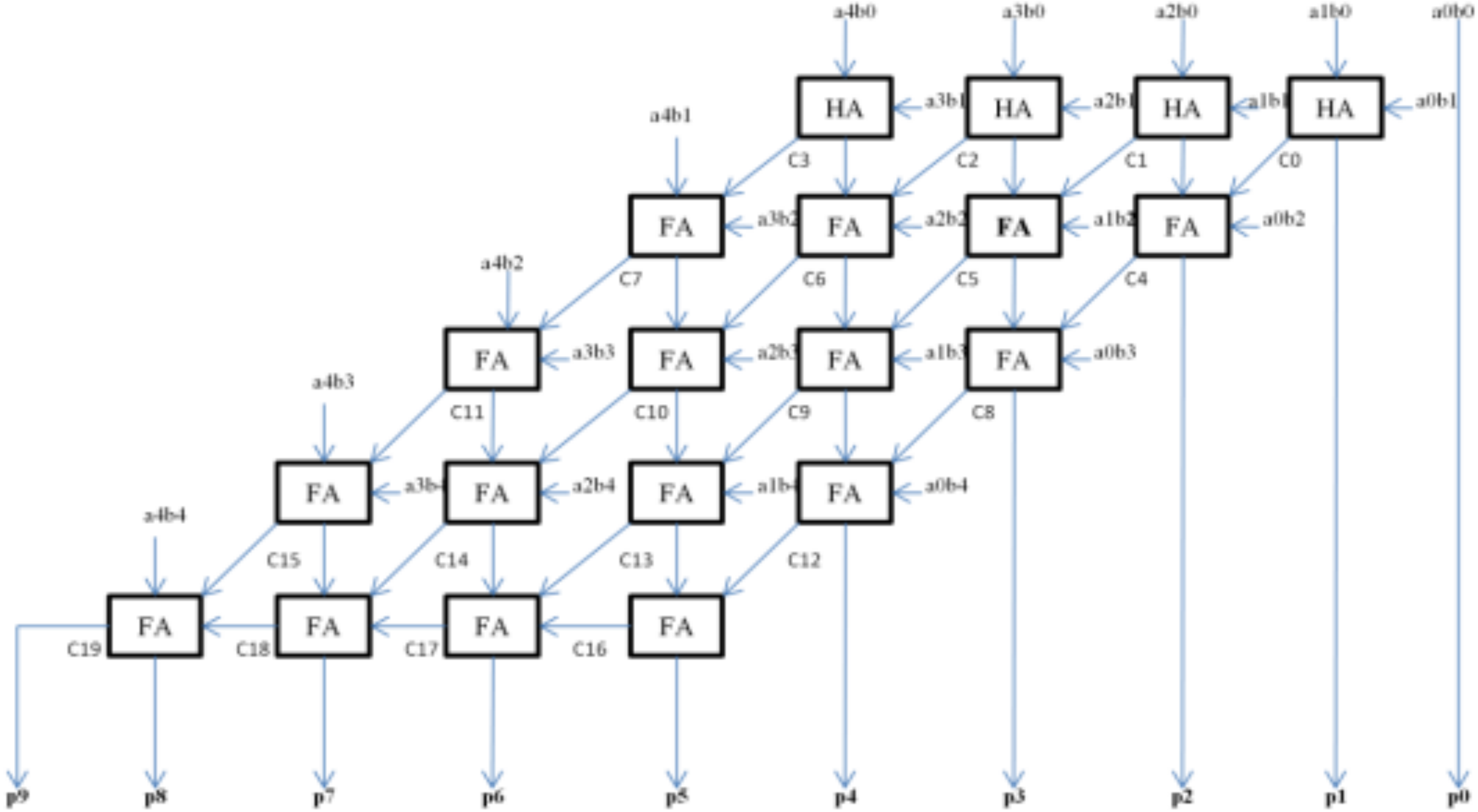

BR register
Sequence counter (SC)
Complementer and parallel adder
AC register
QR register
Q_n
Q_{n+1}

Array multiplier

- Array multiplier uses a grid of full and half adders to perform nearly simultaneous addition of product terms.
- AND gates are used to form these product terms before they are fed into the adder array.
- Unlike sequential multipliers that check bits one at a time, array multipliers form the product bits all at once, making the operation faster.
- The speed advantage comes at a cost of requiring a large number of gates, which became economical with the advent of integrated circuits.



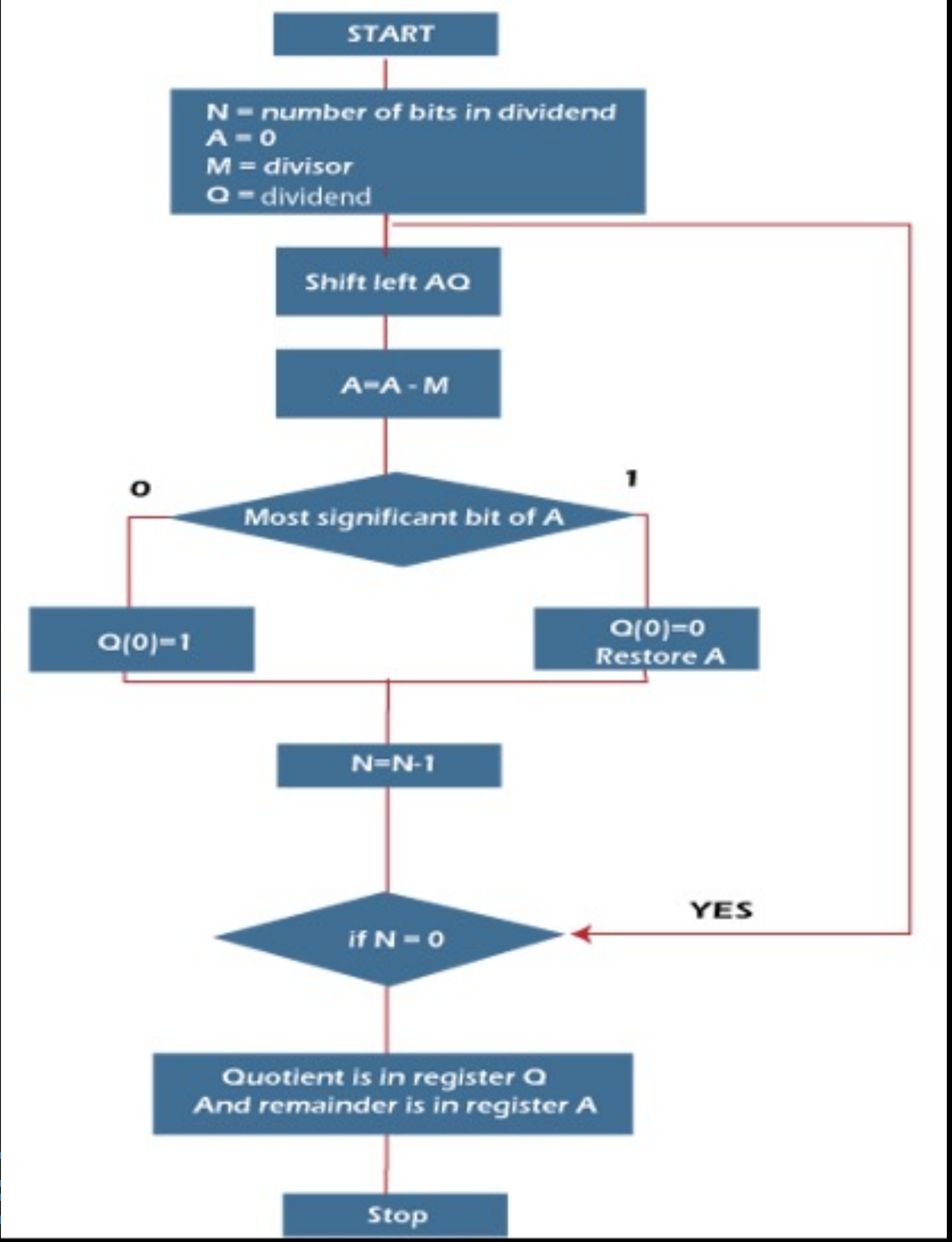
A_3 A_2 A_1 A_0
$+$ B_3 B_2 B_1 B_0
A_3B_0 A_2B_0 A_1B_0 A_0B_0 PP0
A_3B_1 A_2B_1 A_1B_1 A_0B_1 PP1
A_3B_2 A_2B_2 A_1B_2 A_0B_2 PP2
A_3B_3 A_2B_3 A_1B_3 A_0B_3 PP3
P_7 P_6 P_5 P_4 P_3 P_2 P_1 P_0 PP4


a4b0
a3b0
a2b0
a1b0
a0b0
HA
a3b1
HA
a2b1
HA
a1b1
HA
a0b1
a4b1
C3
C2
C1
C0
FA
a3b2
FA
a2b2
FA
a1b2
FA
a0b2
a4b2
C7
C6
C5
C4
FA
a3b3
FA
a2b3
FA
a1b3
FA
a0b3
a4b3
C11
C10
C9
C8
FA
a3b4
FA
a2b4
FA
a1b4
FA
a0b4
a4b4
C15
C14
C13
C12
FA
FA
FA
FA
C19
C18
C17
C16
p9
p8
p7
p6
p5
p4
p3
p2
p1
p0

```
1 1 0 0
0 1 1 0
____________
```

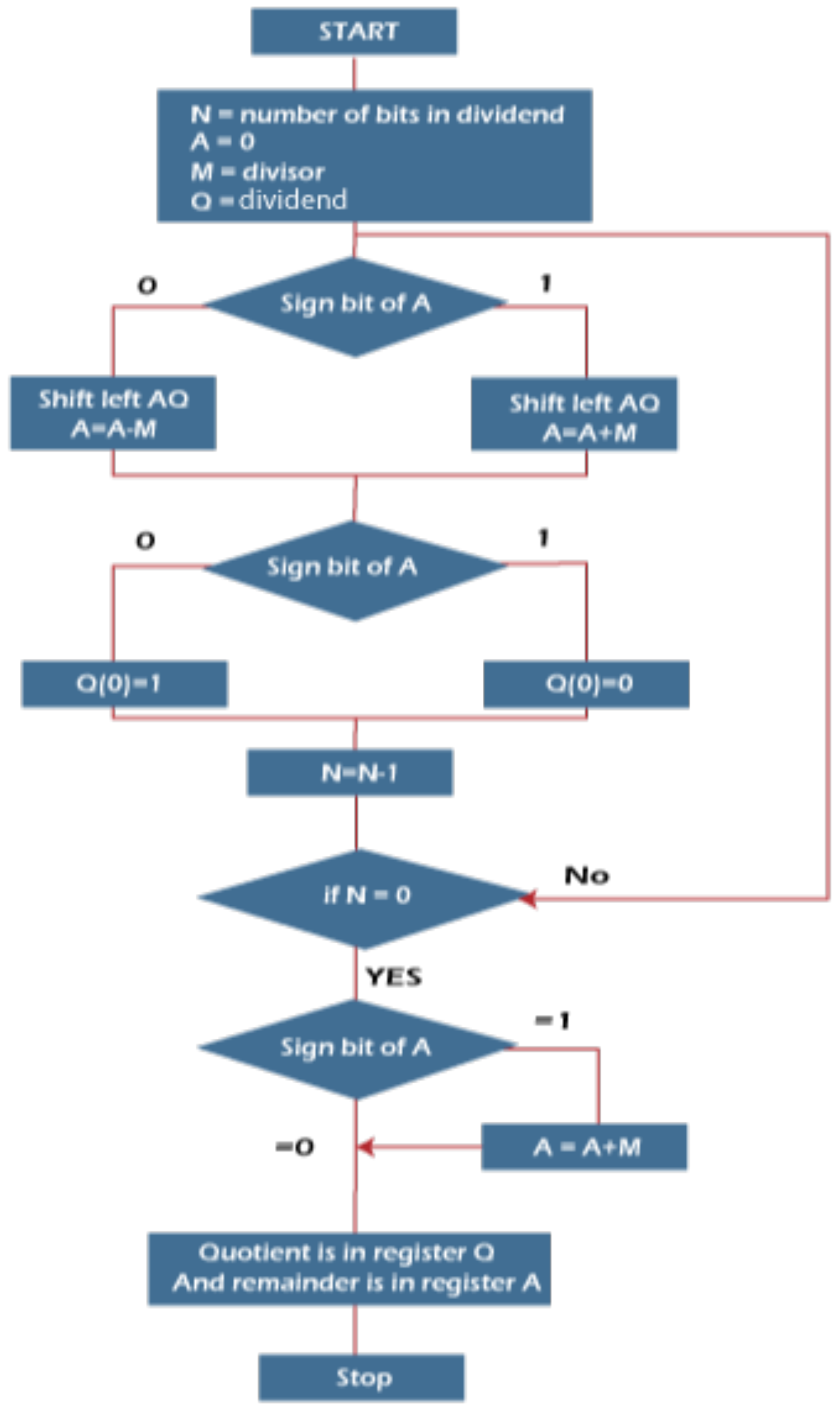
Restoring Division Algorithm

n	M	A	Q	Action/Operation
4	00011	00000	1011	Initialization
		00001	011?	SL AQ
		11110	011?	A = A-M
3	00011	00001	0110	QO ← 0
		00010	110?	SL AQ
		11111	110?	A = A-M
2	00011	00010	1100	QO ← 0
		00101	100?	SL AQ
		00010	100?	A = A-M
1	00011	00010	1001	QO ← 1
		00101	001?	SL AQ
		00010	001?	A = A-M
0	00011	00010	0011	QO ← 1


START
N = number of bits in dividend
A = 0
M = divisor
Q = dividend
Shift left AQ
A=A - M
Most significant bit of A
0
1
Q(0)=1
Q(0)=0
Restore A
N=N-1
if N = 0
YES
Quotient is in register Q
And remainder is in register A
Stop

Non-Restoring Division Algorithm

n	M	A	Q	Action/Operation
4	00011	00000	1011	Initialization
		00001	011?	SL AQ
		11110	011?	A = A-M
3	00011	11110	0110	QO ← 0
		11100	110?	SL AQ
		11111	110?	A = A+M
2	00011	11111	1100	QO ← 0
		11111	100?	SL AQ
		00010	100?	A = A+M
1	00011	00010	1001	QO ← 1
		00101	001?	SL AQ
		00010	001?	A = A-M
0	00011	00010	0011	QO ← 1



Q Add -35 and -31 in binary using 8-bit registers, in signed 1's complement and signed 2's complement?

Floating point representation

- Problem with representation we have already studied is that it do not works well if the number to be stored is either too small or too large, is it take very large amount of memory.
- Imagine a number 6.023 * 10 ^{23} will require around 70 bits to be stored.
- So in scientific application or statics it is a problem to store very small or very large number.

- Floating point representation a special kind of sign magnitude representation.
- Floating point number is stored in mantissa/exponent form i.e. $m*r^e$
- Mantissa is a signed magnitude fraction for most of the cases.
- The exponent is stored in biased form.

- The biased exponent is an unsigned number representing signed true exponent.
- If the biased exponent field contains K bits, the biased = 2^{k-1}
- True value expression is $V = (-1)^S(.M)_2*2^{E-Bias}$, note it is explicit representation
- True value expression is $V = (-1)S(1.M)_2*2^{E-Bias}$, note it is implicit representation, it has more precision than explicit normalization.

- Biased exponent(E) = True exponent(e) + Bias
- Range of true exponent: from -2^{k-1} to $+2^{k-1}-1$, where k is number of bits assigned for exponent
- After adding bias 2^{k-1}, new range go from 0 to $2^{k}-1$

- How to convert a signed number into floating point representation
- The floating-point normalized number distribution is not uniform. Representable number are dense towards zero and spared towards max value
- This uneven distribution result negligible effect of rounding towards zero and dominant towards max value.

Q Represent -21.75 (s=1, k=7, m=8)

IEEE 754 floating point standard

- The **IEEE Standard for Floating-Point Arithmetic** (**IEEE 754**) is a technical standard for floating-point arithmetic 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 portably. Many hardware floating-point units use the IEEE 754 standard.

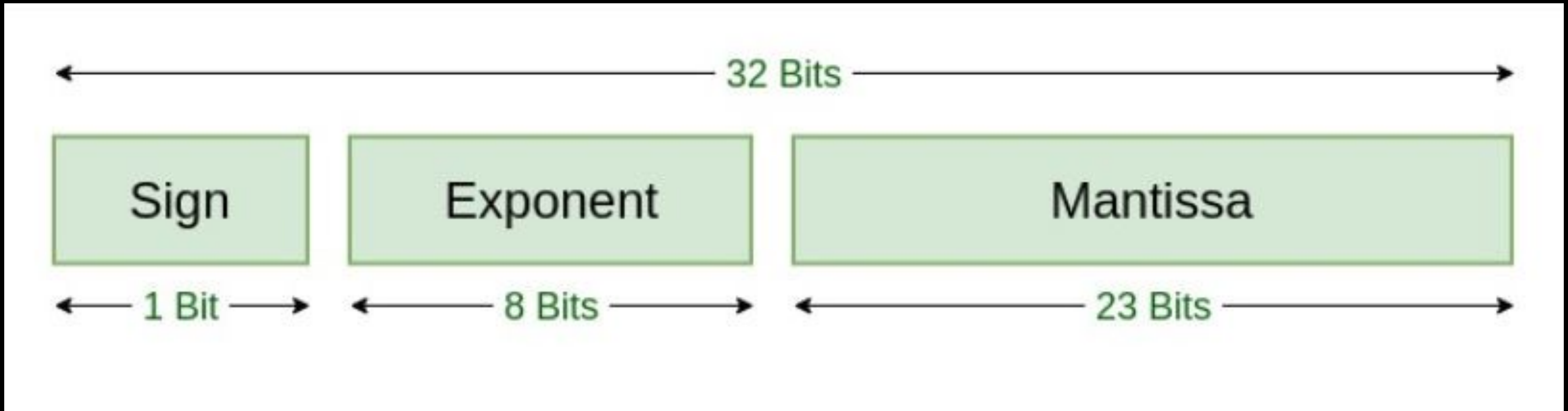
IEEE 754 floating point standard

- IEEE 754 standard represent floating point number standards
- It gives a provision for +-0 & +-∞ (by reserving certain pattern for Mantissa/Exponent pattern)
- there are number of modes for storage from half precision (16 bits) to very lengthy notations
- based of the system is 2
- the floating-point number can be stored either in implicit normalized form or in fractional form
- If the biased exponent field contains K bits, the biased = 2^{k-1} -1
- Certain Mantissa/Exponent pattern does not denote any number (NAN i.e. not a number)

Name	Common name	Mantissa Bits	Exponent bits	Exponent bias	E min	E max
binary16	Half precision	10	5	$2^{5-1}-1 = 15$	−14	+15
binary32	Single precision	23	8	$2^{8-1}-1 = 127$	−126	+127
binary64	Double precision	52	11	$2^{11-1}-1 = 1023$	−1022	+1023
binary128	Quadruple precision	112	15	$2^{15-1}-1 = 16383$	−16382	+16383
binary256	Octuple precision	236	19	$2^{19-1}-1 = 262143$	−262142	+262143

Single pression

Sign bit (1)	Exponent (8)	Mantissa (23)	Value
0	00…….0(E=0)	00…….0(M=0)	+0
1	00…….0(E=0)	00…….0(M=0)	-0
0	11……1(E=255)	00…….0(M=0)	+∞
1	11……1(E=255)	00…….0(M=0)	-∞
0/1	1<=E<=254	XX…….X (M! =0)	Implicit normalised number
0/1	00…….0(E=0)	XX…….X (M! =0)	Fraction
0/1	11……1(E=255)	XX…….X (M! =0)	NAN (Not a Number)



Q Consider the following representation of a number in IEEE 754 single-precision floating point format with a bias of 127.

S : 1   E : 10000001   F : 11110000000000000000000

Here, S,E and F denote the sign, exponent, and fraction components of the floating point representation. The decimal value corresponding to the above representation (rounded to 2 decimal places) is ____________.

Q Given the following binary number in 32-bit (single precision) IEEE-754 format:

00111110011011010000000000000000

The decimal value closest to this floating-point number is

Q The decimal value 0.5 in IEEE single precision floating point representation has

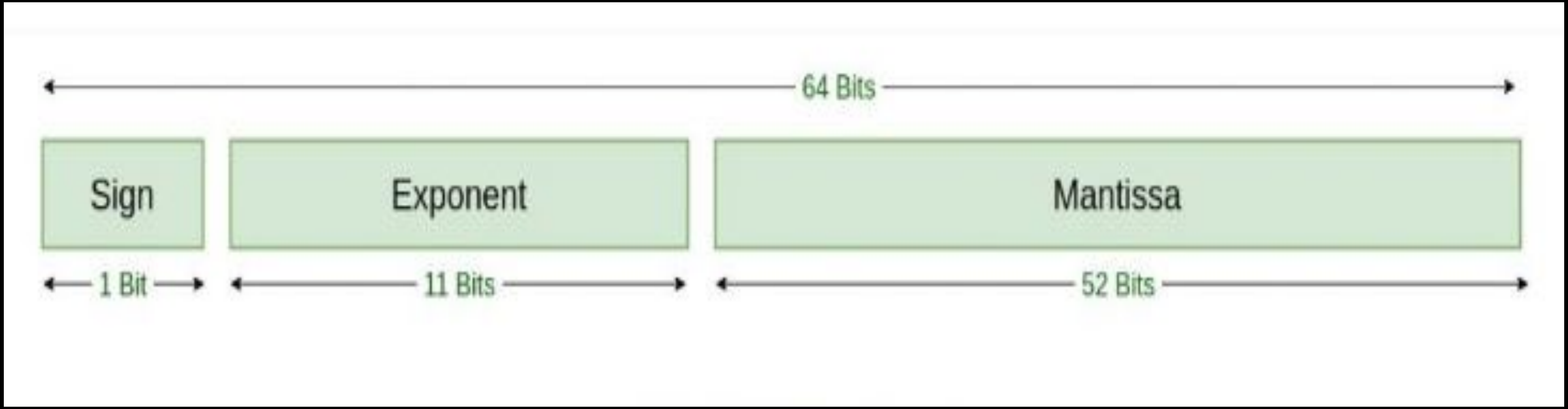
a) fraction bits of 000…000 and exponent value of 0

b) fraction bits of 000…000 and exponent value of −1

c) fraction bits of 100…000 and exponent value of 0

d) no exact representation

Double Precision


64 Bits
Sign
Exponent
Mantissa
1 Bit
11 Bits
52 Bits