Unit 2

18CSC203J-Computer Organization and Architecture

Course Outcome

- **CLR-2:Analyze the functions of arithmetic *units like adders, multipliers etc.***

- ***CLR-6:Simulate simple fundamental units like half adder, full adder etc.***

- **CLO-2:*Apply Boolean algebra as related in designing computer logic through simple combinational and sequential logic circuits***

Table of Contents

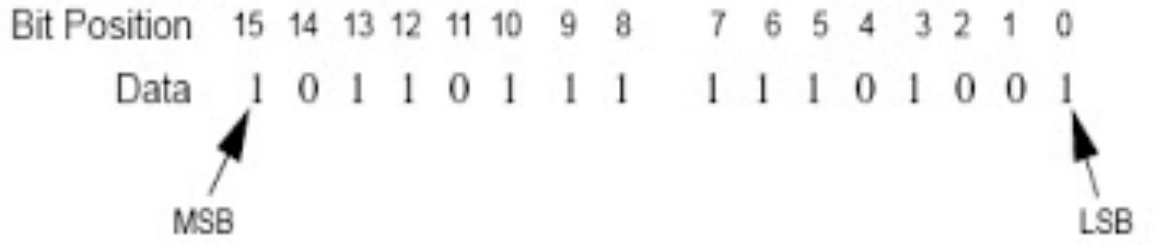
- Addition and subtraction of signed numbers
- Design of fast adders – Ripple Carry Adder and Carry Look ahead Adder
- Multiplication of positive numbers
- Signed operand multiplication
- Fast multiplication - Bit pair recoding of Multipliers
- Carry Save Addition of Summands
- Integer division – Restoring Division and Non Restoring Division
- Floating point numbers and operations

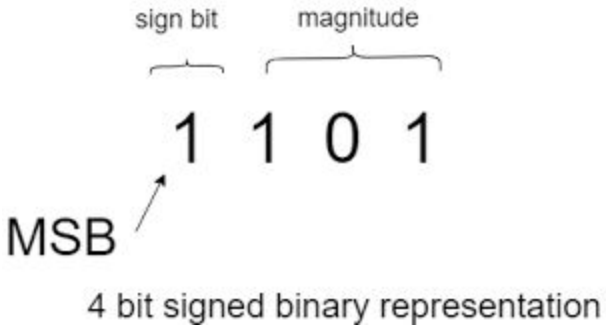
Integer Representation

- Only have 0 & 1 to represent everything
- Positive numbers stored in binary
- e.g. 41=00101001
- No minus sign
- No period
- Sign-Magnitude
- One's Complement
- Two’s compliment

- In integer
- MSB-Most significant Bit
- LSB-Least Significant Bit

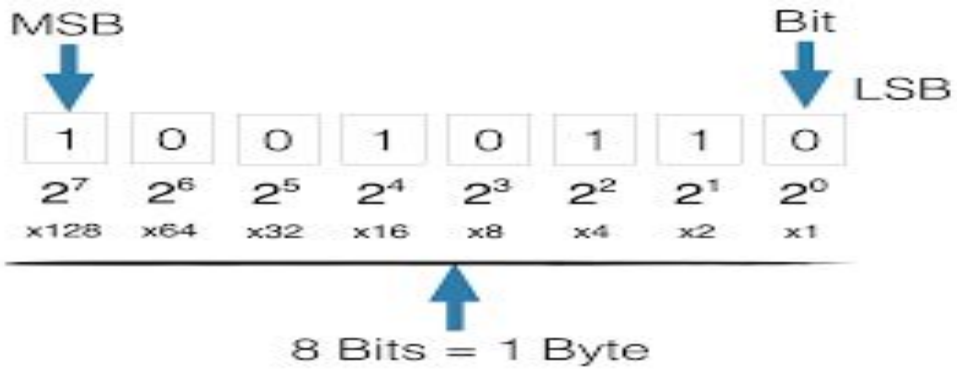
Most Significant Bit (MSB)	This bit has the highest value (greatest weight) and is located at the far left of the bit string.
Least Significant Bit (LSB)	This bit has the lowest value (bit position zero) and is located at the far right of the bit string.





4 bit signed binary representation

If MSB is 0 then the integer is +ve number
MSB is 1 then the integer is –ve number



The value of bits in signed and unsigned binary numbers

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Unsigned	27 = 128	26 = 64	25 = 32	24= 16	23= 8	22 = 4	21 = 2	20 = 1
Signed	-(27) = -128	26 = 64	25 = 32	24 = 16	23 = 8	22 = 4	21= 2	20 = 1

Integer Representation (cont'd)

- Sign Magnitude: One's Complement Two's Complement

000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- Issues: balance, number of zeros, ease of operations
- Which one is best? Why?

SUMMARY OF THE TABLE

- SIGN & MAGNITUE SYSTEM: Negative value is obtained by changing the sign bit (MSB)
- SIGNED 1'S COMPLEMENT: Negative number is obtained by complementing each bit of the corresponding positive number i.e (2n-1) –N
- SIGNED 2'S COMPLEMENT: Negative number is obtained by taking 2's complement of positive number
  - 2^n-N
  - Range: - (2^{n-1}) to + (2^{n-1}-1)

Range of Values

$n = 8$

i.e. 8 bit number

1's Complement $(2^n - 1) - N$

2's complement $-(2^{n-1})$ to $(2^{n-1} - 1)$

Example

$n = 8$

$-(2^{8-1}) = -2^7 = -128$ minimum

$+(2^{8-1}) - 1 = 2^7 - 1 = +127$ Maximum

* BINARY ARITHMETIC

* for all digital Computer and digital system
* Binary addition, Subtraction, Multiplication and division.

Binary Addition:

Truth table

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

Example :

$$\begin{array}{r} 2 \\ +\ 3 \\ \hline 5 \end{array} \qquad \begin{array}{r} 8\ 4\ 2\ 1 \\ 0\ 0\ 1\ 0 \\ 0\ 0\ 1\ 1 \\ \hline 0\ 1\ 0\ 1 \end{array} = 5$$

$$\begin{array}{r} +\ 9 \\ 22 \\ \hline 31 \end{array} \qquad \begin{array}{r} 16\ 8\ 4\ 2\ 1 \\ 0\ 1\ 0\ 0\ 1 \\ 1\ 0\ 1\ 1\ 0 \\ \hline 1\ 1\ 1\ 1\ 1 \end{array} = 31$$

Binary Subtraction

- Subtraction and Borrow, these two words will be used very frequently for the binary subtraction. There four rules of the binary subtraction.

A	B	BORROW	DIFFERENCE
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

0011010 - 001100 = 00001110

1 1 borrow

0 0 ~~1 1~~ 0 1 0 = 26_{10}

- 0 0 0 1 1 0 0 = 12_{10}

0 0 0 1 1 1 0 = 14_{10}

Binary Subtraction

A	B	Borrow	Difference
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

By Using 1's & 2's Complement

Ex:

$$\begin{array}{r} -5 \\ -2 \\ \hline -7 \end{array}$$

+5 : 0101 $\xrightarrow{1's}$ 1010

+2 = 0010 $\longrightarrow$ 1101

$$\begin{array}{r} 1010 \\ +1 \\ \hline 1011 \end{array} \rightarrow -5 \qquad \begin{array}{r} 1101 \\ +1 \\ \hline 1110 \end{array} \rightarrow -2$$

$$\begin{array}{r} 1011 \\ 1110 \\ \hline 1)1001 \end{array}$$

Carry $\quad \underline{1}001$

MSB is -ve take 2's

$\underline{1}001 \rightarrow 0110$ (1)

$\quad 0111$

$-(7)$

$2-4 \Rightarrow 2+(-4) = -2$ +4 $\Rightarrow$ 0100

$= 1011$ (1)

$\quad 1100$

$$\begin{array}{r} 0010 \\ +1100 \\ \hline 1110 \end{array} \qquad \begin{array}{r} 0001 \\ \hline 0010 = 2 \end{array}$$

$-$

$\therefore -2$

2 - 4

$$\begin{array}{r} 0010 \\ 0100 \\ \hline 1110 \end{array} \qquad -0010$$

(-2)

5 - 6

$$\begin{array}{r} 0101 \\ 0110 \\ \hline 1111 \end{array} \qquad \begin{array}{r} 0000 \\ +1 \\ \hline 0001 = 1 \end{array}$$

$\therefore -1$

Binary Multiplication

- Binary multiplication is similar to decimal multiplication. It is simpler than decimal multiplication because only 0s and 1s are involved. There four rules of the binary multiplication.

Case	A x B	Multiplication
1	0 x 0	0
2	0 x 1	0
3	1 x 0	0
4	1 x 1	1

Example:

0011010 x 001100 = 100111000

$$
\begin{array}{rl}
0\,0\,1\,1\,0\,1\,0 & = 26_{10} \\
\times\, 0\,0\,0\,1\,1\,0\,0 & = 12_{10} \\
\hline
0\,0\,0\,0\,0\,0\,0 & \\
0\,0\,0\,0\,0\,0\,0 & \\
0\,0\,1\,1\,0\,1\,0 & \\
0\,0\,1\,1\,0\,1\,0 & \\
\hline
0\,1\,0\,0\,1\,1\,1\,0\,0\,0 & = 312_{10}
\end{array}
$$

Binary Division

- Binary division is similar to decimal division. It is called as the long division procedure

101010 / 000110 = 000111

```
                 111      = 7₁₀
          ------------
000110 ) 101010   = 42₁₀
         -110     = 6₁₀
         ----
          1001
          -110
          ----
           110
           110
          ----
             0
```

Half Adder

- Half adder is a combinational logic circuit with two input and two output. The half adder circuit is designed to add two single bit binary number A and B. It is the basic building block for addition of two single bit numbers. This circuit has two outputs carry and sum.

Truth Table and Circuit Diagram

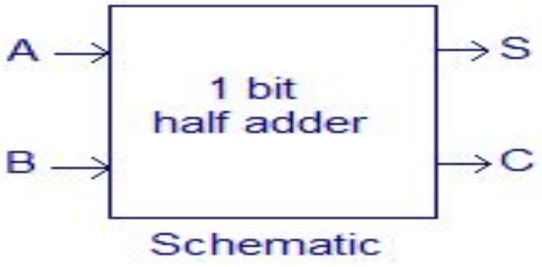
Inputs		Outputs	
A	B	S	C
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

Truth table

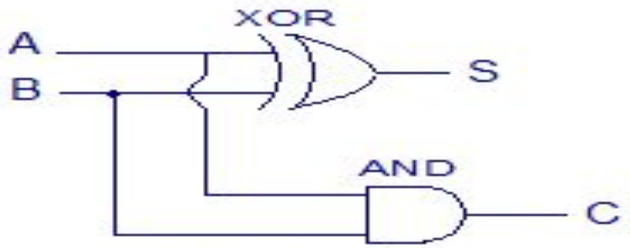
$$\text{SUM } S = A.\overline{B} + \overline{A}.B$$

$$\text{CARRY } C = A.B$$

Boolean Expression



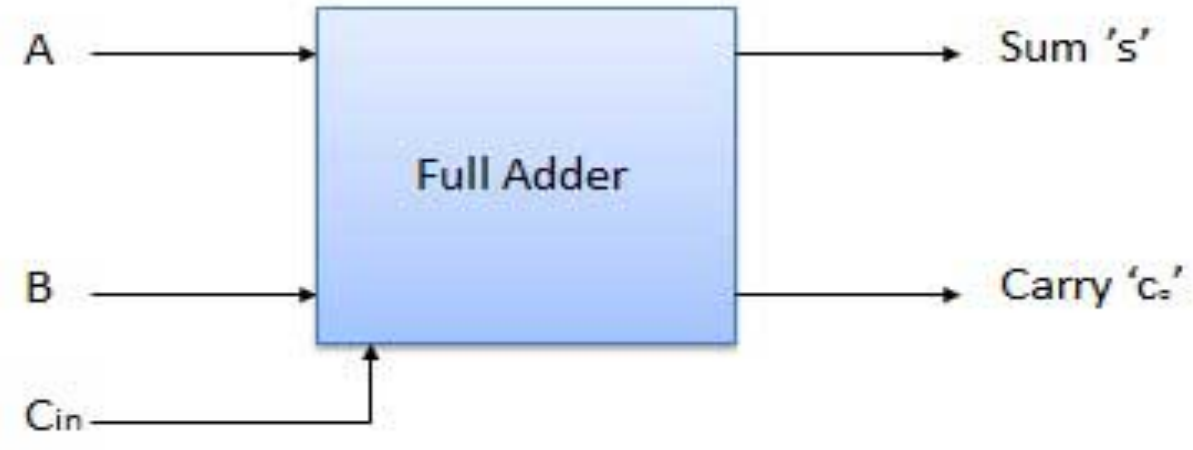
Schematic



Realization

Full Adder

- Full adder is developed to overcome the drawback of Half Adder circuit. It can add two one-bit numbers A and B, and carry c. The full adder is a three input and two output combinational circuit.

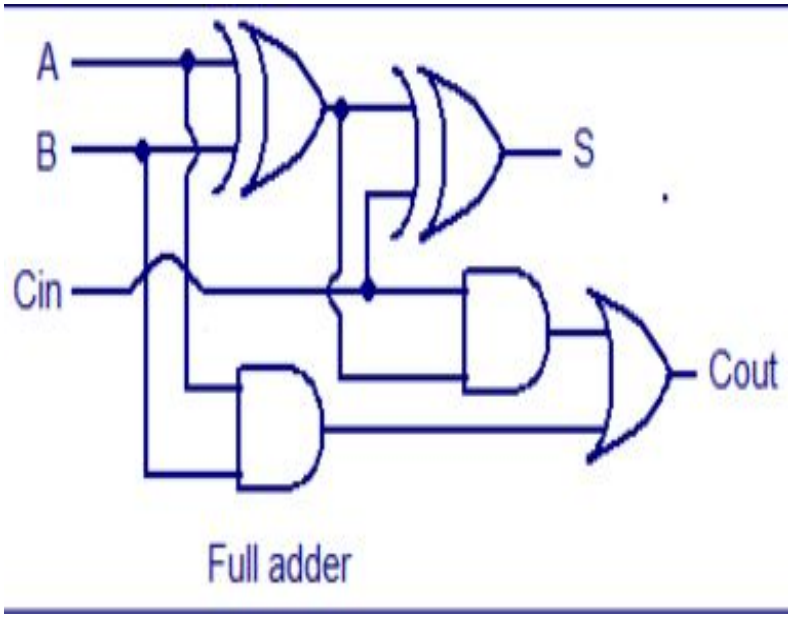


Truth Table and Circuit Diagram

$$S = A\overline{B}\overline{C} + \overline{A}\,\overline{B}C + ABC + \overline{A}B\overline{C}$$
$$= C\,(AB + \overline{A}\,\overline{B}) + \overline{C}\,(\overline{A}B + A\,\overline{B})$$
$$= C\,\overline{(\overline{A}B + A\,\overline{B})} + \overline{C}\,(\overline{A}B + A\,\overline{B})$$
$$= C\,\overline{(A \oplus B)} + \overline{C}\,(A \oplus B) \quad = A \oplus B \oplus C$$

$$C_{out} = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$
$$= (\overline{A}B + A\overline{B})\,C + AB\,(\overline{C} + C)$$
$$= (A \oplus B).\,C + AB.$$

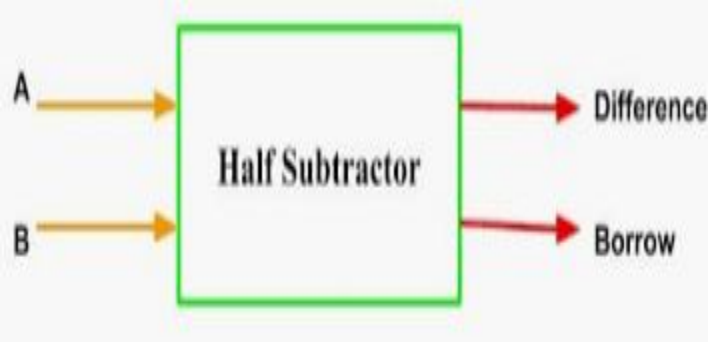
Inputs			Output	
A	B	Cin	S	Co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Full adder

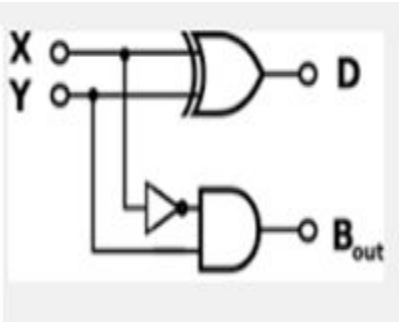
Half Subtractor

- Half subtractor is a combination circuit with two inputs and two outputs (difference and borrow). It produces the difference between the two binary bits at the input and also produces a output (Borrow) to indicate if a 1 has been borrowed. In the subtraction (A-B), A is called as Minuend bit and B is called as Subtrahend bit.



Truth Table and Circuit Diagram

A	B	BORROW	DIFFERENCE
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

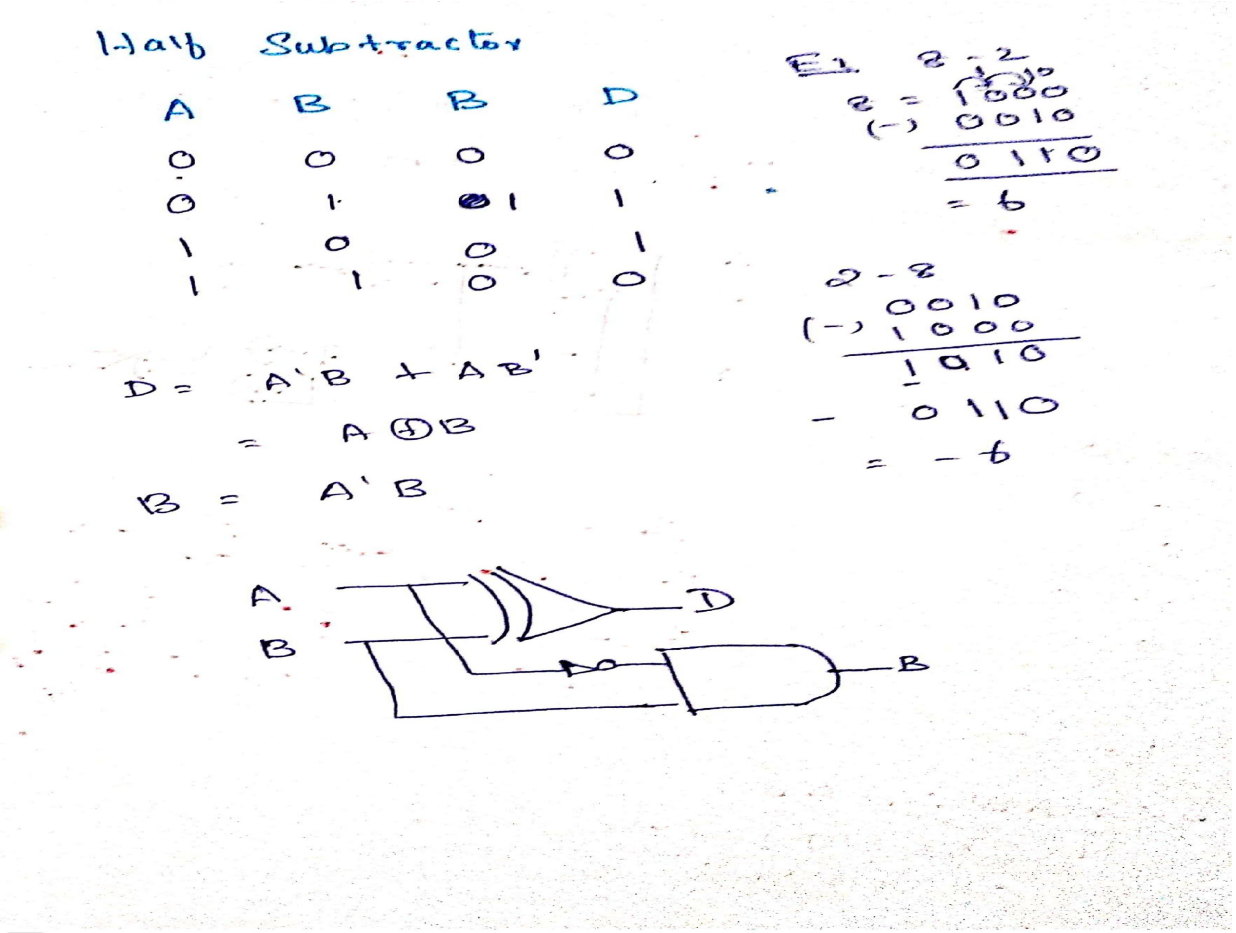


From the truth table, Boolean Expression can be derived as:

$D = A'B + AB' = A \oplus B$

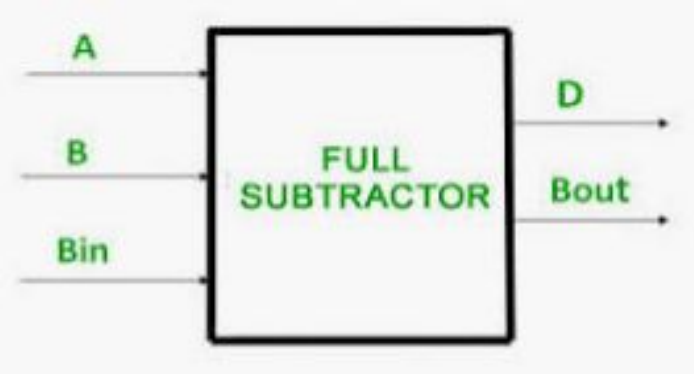
$B_o = A'B$

Half Subtractors



Full Subtractor

- The disadvantage of a half subtractor is overcome by full subtractor. The full subtractor is a combinational circuit with three inputs A, B, Bin and two output D and Bo. A is the minuend, B is subtrahend, Bin is the borrow produced by the previous stage, D is the difference output and Bo is the borrow output.



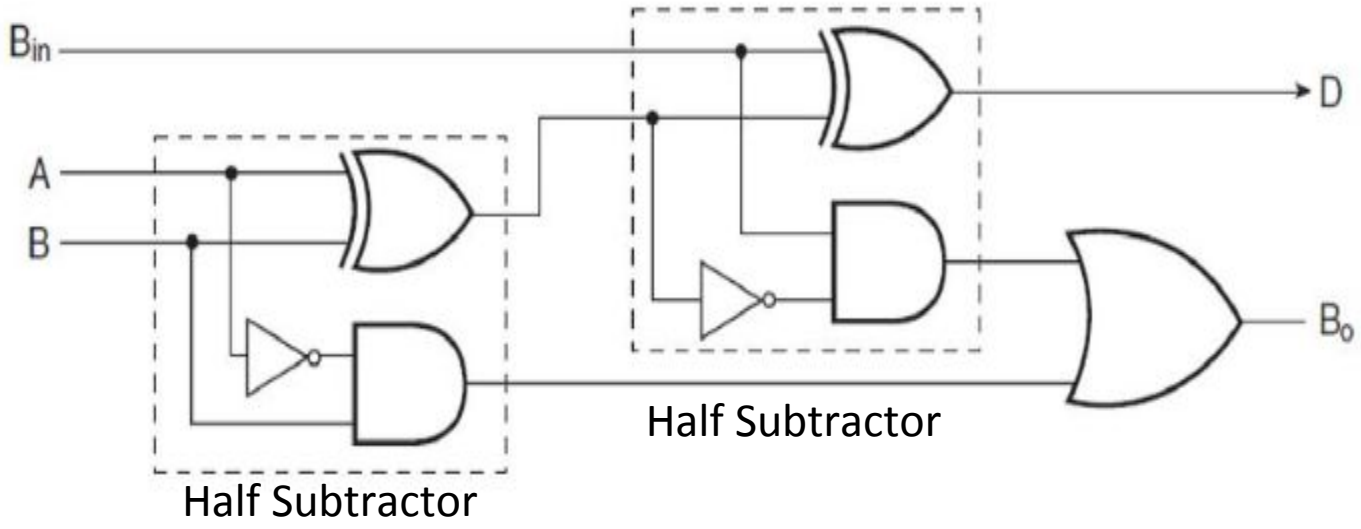
INPUT			OUTPUT	
A	B	Bin	D	Bout
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

D_i

$$A'B'B_{in} + A'BB_{in}' + AB'B_{in}' + ABB_{in}$$
$$= A'(B'B_{in} + BB_{in}') + A(B' B_{in}' + BB_{in})$$
$$= A'(B \oplus B_{in}) + A(B \oplus Bin)'$$
$$= A \oplus B \oplus B_{in}$$

B_0

$$A'B'B_{in} + A'BB_{in}' + A'BB_{in} + ABB_{in}$$
$$= A'B'B_{in} + ABB_{in} + A'B(B_{in} + B_{in}')$$
$$= B_{in}(A \oplus B)' + A'B$$



Ripple Carry Adder

Typical Ripple Carry Addition is a Serial Process:

- Addition starts by adding LSBs of the augend and addend.
- Then next position bits of augend and addend are added along with the carry (if any) from the preceding bit.
- This process is repeated until the addition of MSBs is completed.
- Speed of a ripple adder is limited due to carry propagation or carry ripple.
- Sum of MSB depends on the carry generated by LSB.

Ripple Carry Adder

Example: 4-bit Carry Ripple Adder

- Assume to add two operands A and B where

A = A3 A2 A1 A0

B = B3 B2 B1 B0

A= 1 0 1 1 +

B = 1 1 0 1

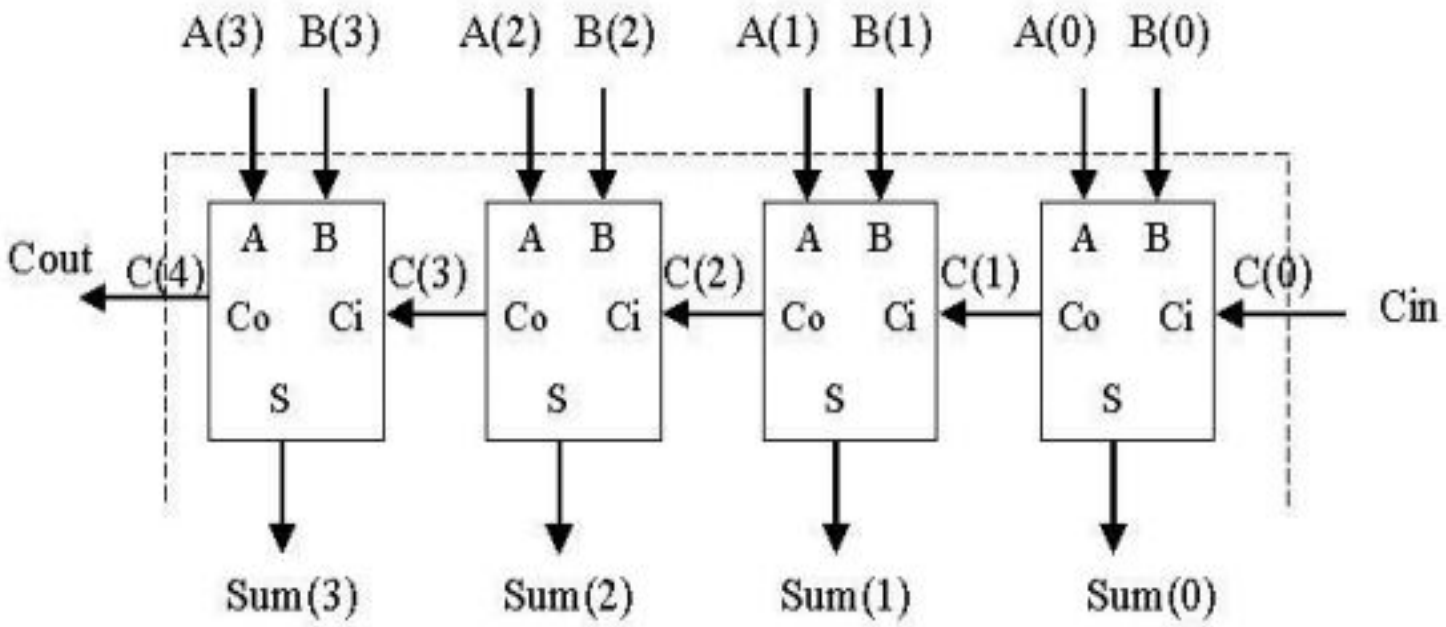
A+B=1 1 0 0 0

Cout S3 S2 S1 S0

Ripple Carry Adder

Carry Propagation

- From the above example it can be seen that we are adding 3 bits at a time sequentially until all bits are added.
- A full adder is a combinational circuit that performs the arithmetic sum of three input bits: augends Ai, addend Bi and carry in Cin from the previous adder.
- Its result contain the sum Si and the carry out, Cout to the next stage.


A(3) B(3)
A(2) B(2)
A(1) B(1)
A(0) B(0)
Cout
C(4)
C(3)
C(2)
C(1)
C(0)
Cin
A
B
Co
Ci
S
Sum(3)
Sum(2)
Sum(1)
Sum(0)

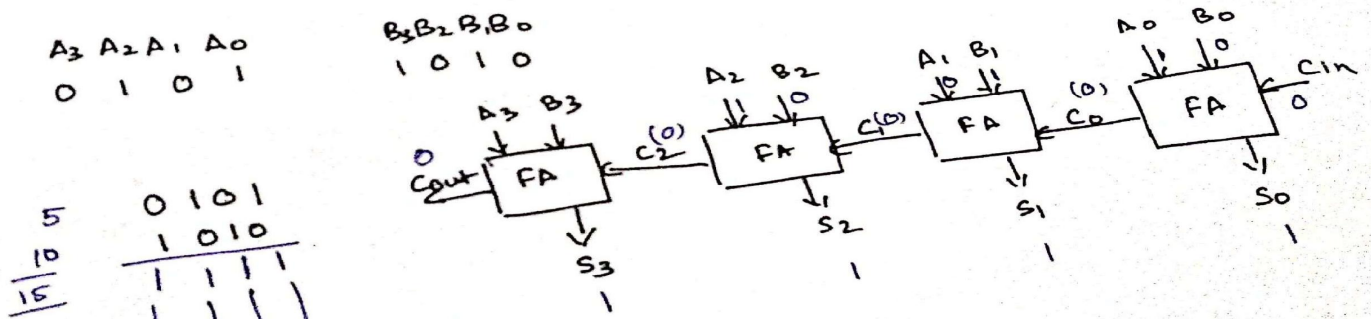
example

Ripple Carry Adder

It is useful for performing Multibit addition

E1

A = 0101 , B = 1010 Cin = 0 (Intial Condition)



5 0101

10 1010

15 1111

S3 S2 S1 S0

C3=0 C2=0 C1=0 C0=0

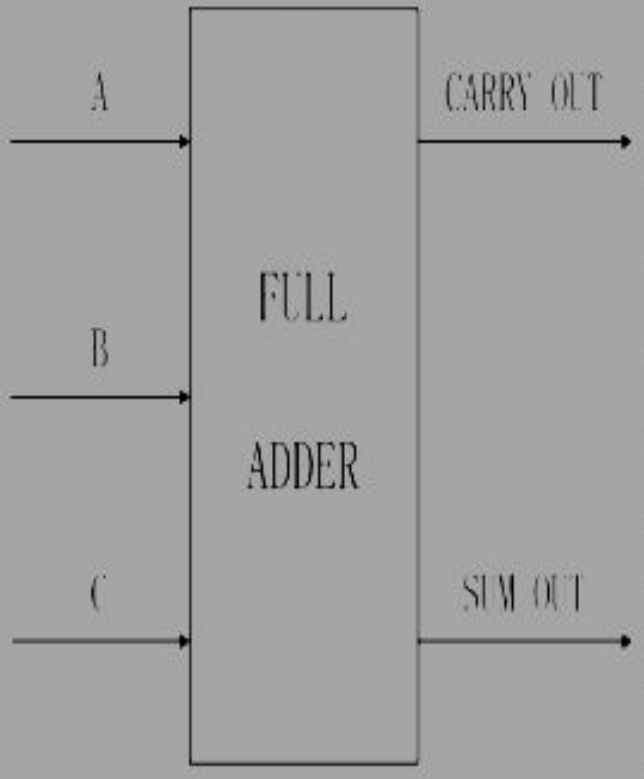
Ripple Carry Adder

4-bit Adder

- A 4-bit adder circuit can be designed by first designing the 1-bit full adder and then connecting the four 1-bit full adders to get the 4-bit adder as shown in the diagram above.
- For the 1-bit full adder, the design begins by drawing the Truth Table for the three input and the corresponding output SUM and CARRY.
- The Boolean Expression describing the binary adder circuit is then deduced.
- The binary full adder is a three input combinational circuit which satisfies the truth table given below.

Ripple Carry Adder

Full Adder

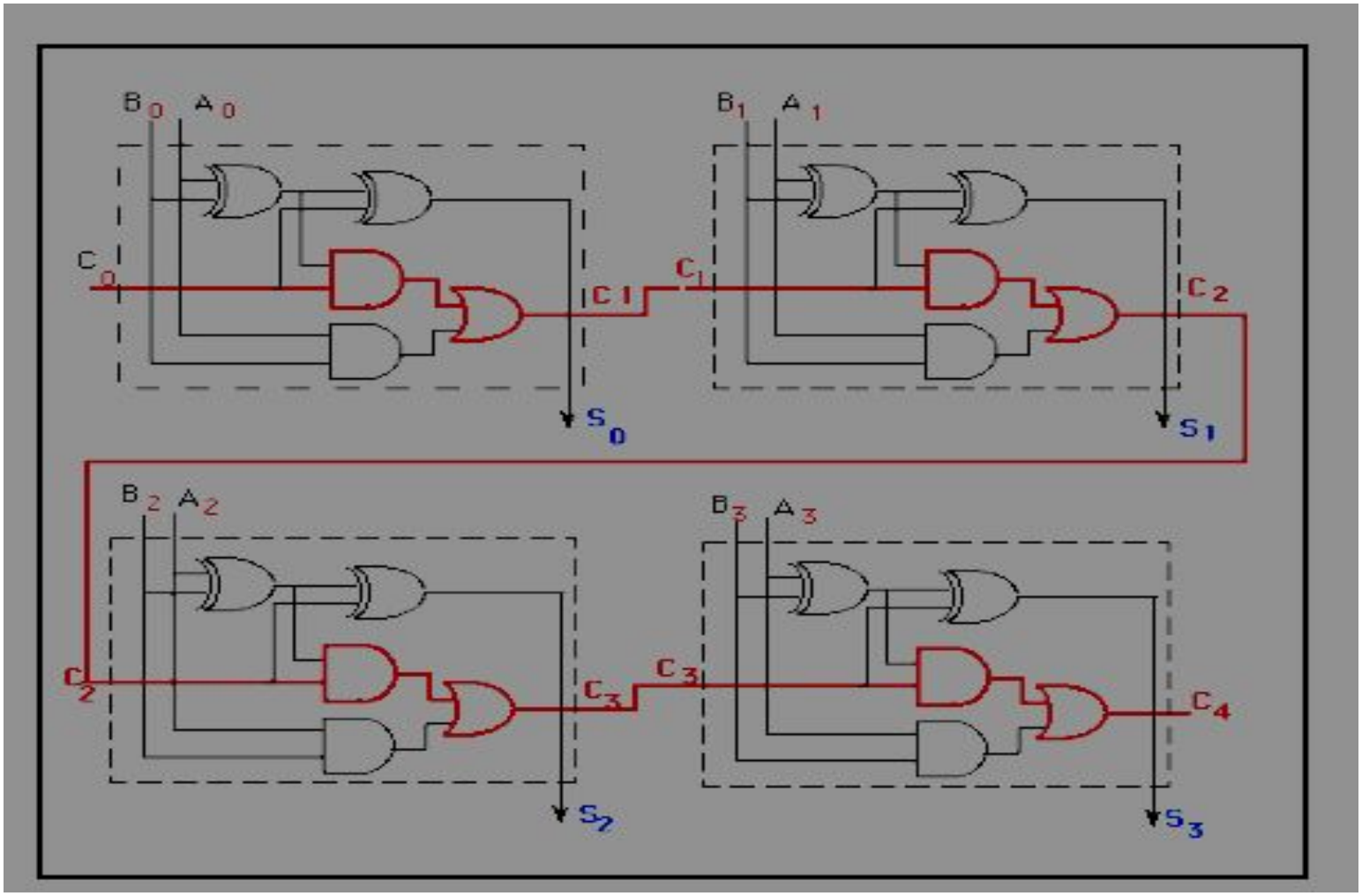


A	B	C	SUM OUT	CARRY OUT
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Fig.2. Diagram and Truth Table of Full Adder

Ripple Carry Adder

4-bit Adder



Draw back of Ripple Carry adder:

* bit Oreration in Serial manner.
* So more delay due to carry Propagation

To reduce the delay:

Carry look ahead Adder (fast adder)

Consider full adder Sum and Carry

$$S = A \oplus B \oplus C$$

$$C = C(A \oplus B) + AB$$

CLA each full adder is going to generate the carry Simultaneously

ie if in full adder any carry is identified then other carry will get generated.

for doing this two Variables are important.

Pi → carry propagate

ci → carry generate

Design of Fast adder: Carry Look-ahead Adder

- A carry-look ahead adders (CLA) is a type of adder used in digital logic. A carry-look ahead adder improves speed by reducing the amount of time required to determine carry bits.
- It can be contrasted with the simpler, but usually slower, ripple carry adder for which the carry bit is calculated alongside the sum bit, and each bit must wait until the previous carry has been calculated to begin calculating its own result and carry bits (see adder for detail on ripple carry adders)

- The carry-look ahead adder calculates one or more carry bits before the sum, which reduces the wait time to calculate the result of the larger value bits.
- In a ripple adder the delay of each adder is 10 ns , then for 4 adders the delay will be 40 ns.
- To overcome this delay Carry Look-ahead Adder is used.

Carry Look-ahead Adder

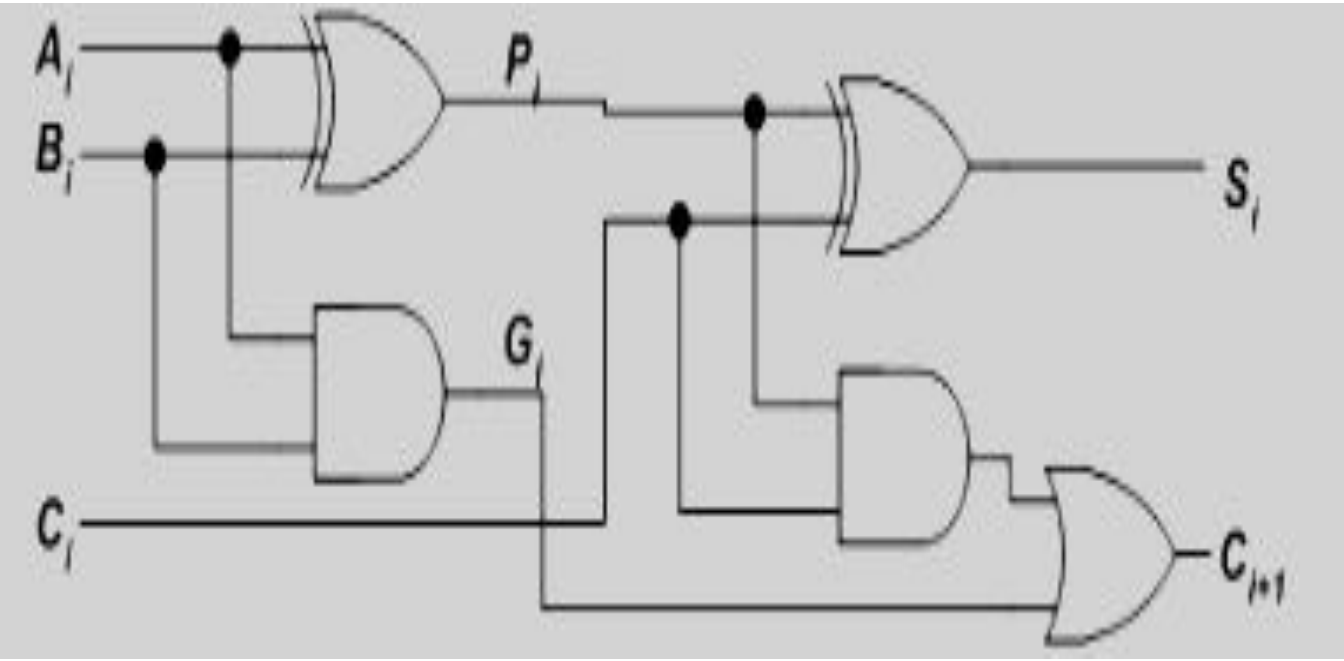
- Different logic design approaches have been employed to overcome the carry propagation delay problem of adders.
- One widely used approach employs the principle of carry look-ahead solves this problem by calculating the carry signals in advance, based on the input signals.
- This type of adder circuit is called as carry look-ahead adder (CLA adder). A carry signal will be generated in two cases:

  (1) when both bits Ai and Bi are 1, or

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

Carry Look-ahead Adder

The Figure shows the full adder circuit used to add the operand bits in the I^{th} column; namely Ai & Bi and the carry bit coming from the previous column (Ci).



Carry Look-ahead Adder

In this circuit, the 2 internal signals Pi and Gi are given by:

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

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

The output sum and carry can be defined as

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

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

Carry Look-ahead Adder

- Gi is known as the carry Generate signal since a carry (Ci+1) is generated whenever Gi =1, regardless of the input carry (Ci).
- Pi is known as the carry propagate signal since whenever Pi =1, the input carry is propagated to the output carry, i.e., Ci+1. = Ci (note that whenever Pi =1, Gi =0).
- Computing the values of Pi and Gi only depend on the input operand bits (Ai & Bi) as clear from the Figure and equations.
- Thus, these signals settle to their steady-state value after the propagation through their respective gates.

Carry Look-ahead Adder

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

$$C_1 = G_0 + P_0C_0$$

$$C_2 = G_1 + P_1C_1 = G_1 + P_1 (G_0 + P_0C_0)$$

$$= G_1 + P_1G_0 + P_1P_0C_0$$

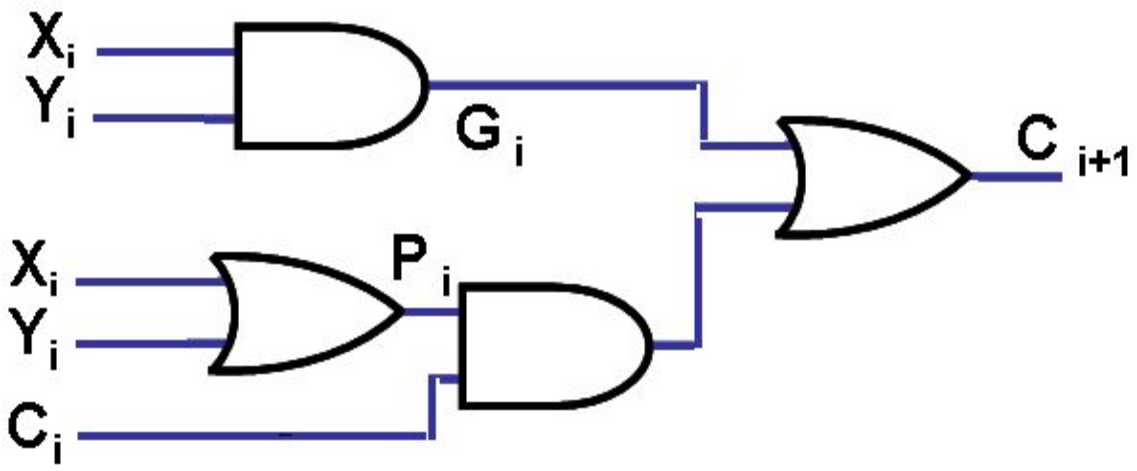
$$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$$

$$C_4 = G_3 + P_3C_3$$

$$= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$$

Carry Look-ahead Adder

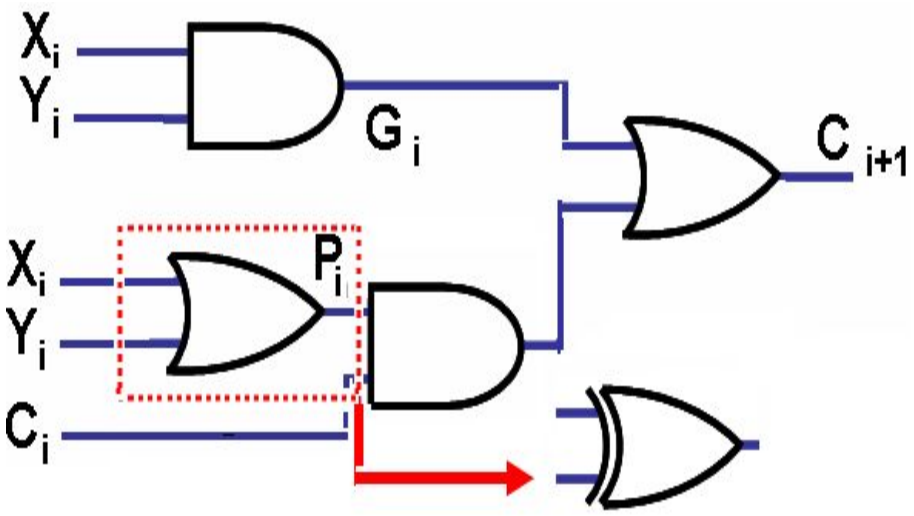
$$\text{Carry } C_{i+1} = X_i Y_i + Y_i C_i + C_i X_i$$
$$= X_i Y_i + C_i (X_i + Y_i)$$
$$= G_i + C_i P_i \text{. (Generate Carry + } C_i \text{ * Propagate Carry)}$$



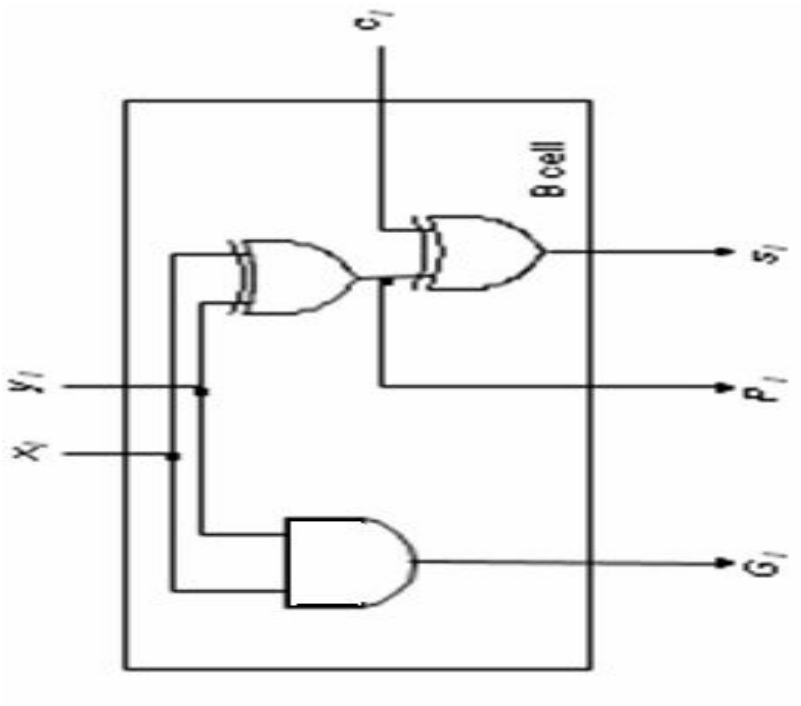
$$\text{Carry } C_{i+1} = G_i + C_i P_i$$
$$\text{i.e. } C_i = (G_{i-1} + P_{i-1} C_{i-1}) \,\&$$
$$C_{i-1} = (G_{i-2} + P_{i-2} C_{i-2}).$$

Carry Look-ahead Adder

- $C_{i+1} = G_i + C_i P_i$.

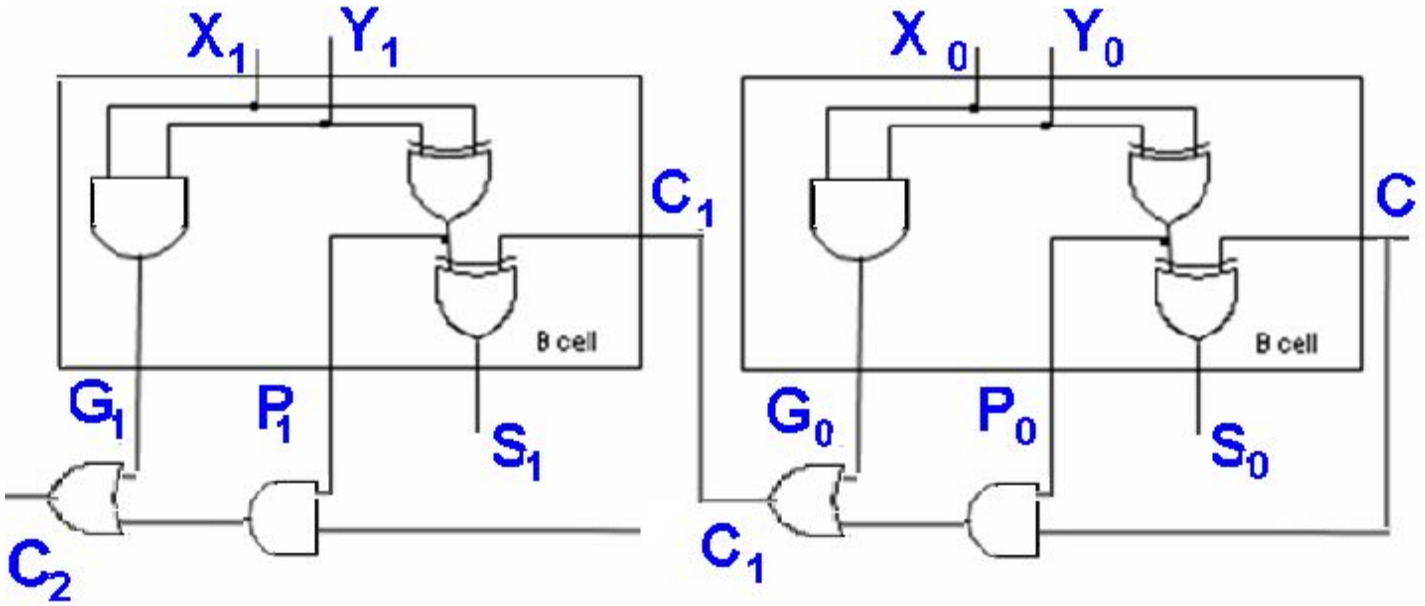


always, except when X_i = 1 & Y_i = 1. But, then Gi = 1to make C_{i+1} = 1; hence Bit cell

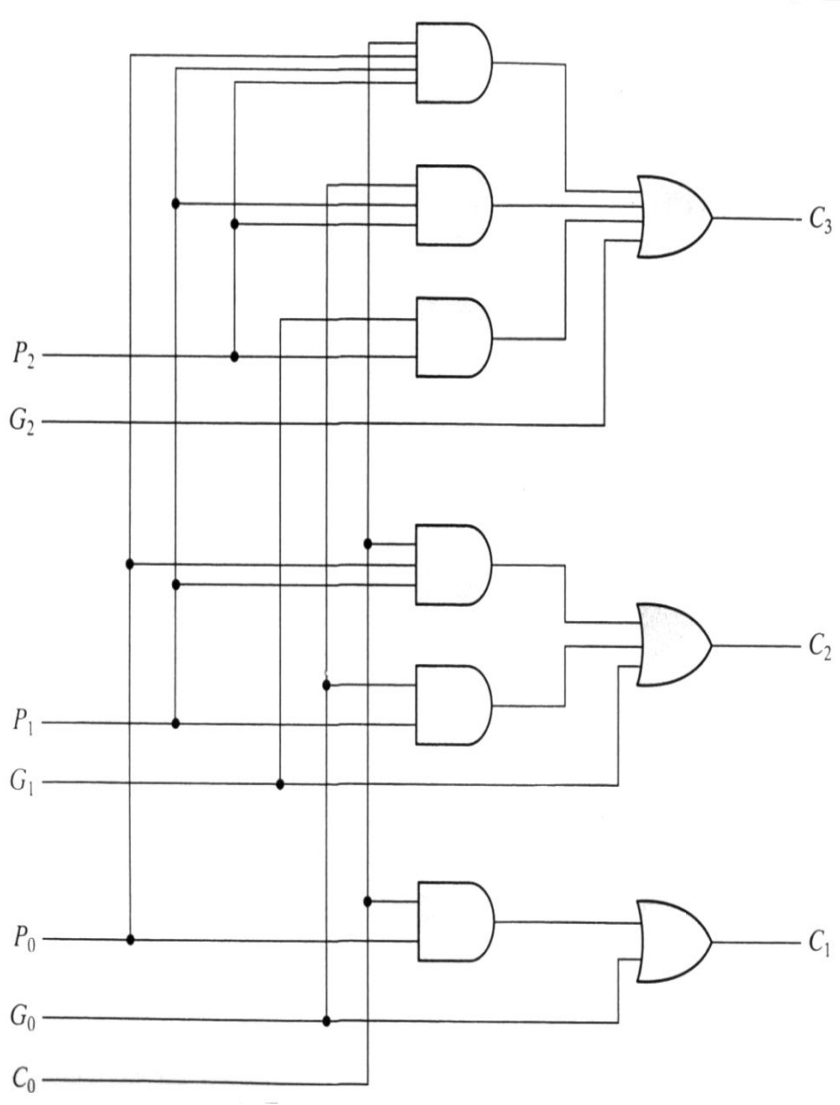


Carry Look-ahead Adder

- In general $C_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \ldots + P_i P_{i-1} \ldots\ldots\ldots P_1 G_0 + P_i P_{i-1} \ldots P_0 G_0$.
- C_{i+1}-- in 3 Gates delay; S_i – in 4 Gates delay irrespective of n. C_1 in 3 gates delay,C_2 in 3 gates delay,C_3 in 3 gates delay and so on.
- S_0 in 4 gates delay, S_1 in 4 gates delay, S_2 in 4 gates delay and so on.



Carry Look-ahead Adder



Implementing these expressions (for a 4-bit adder), results in the logic diagram. (IC- 74182)

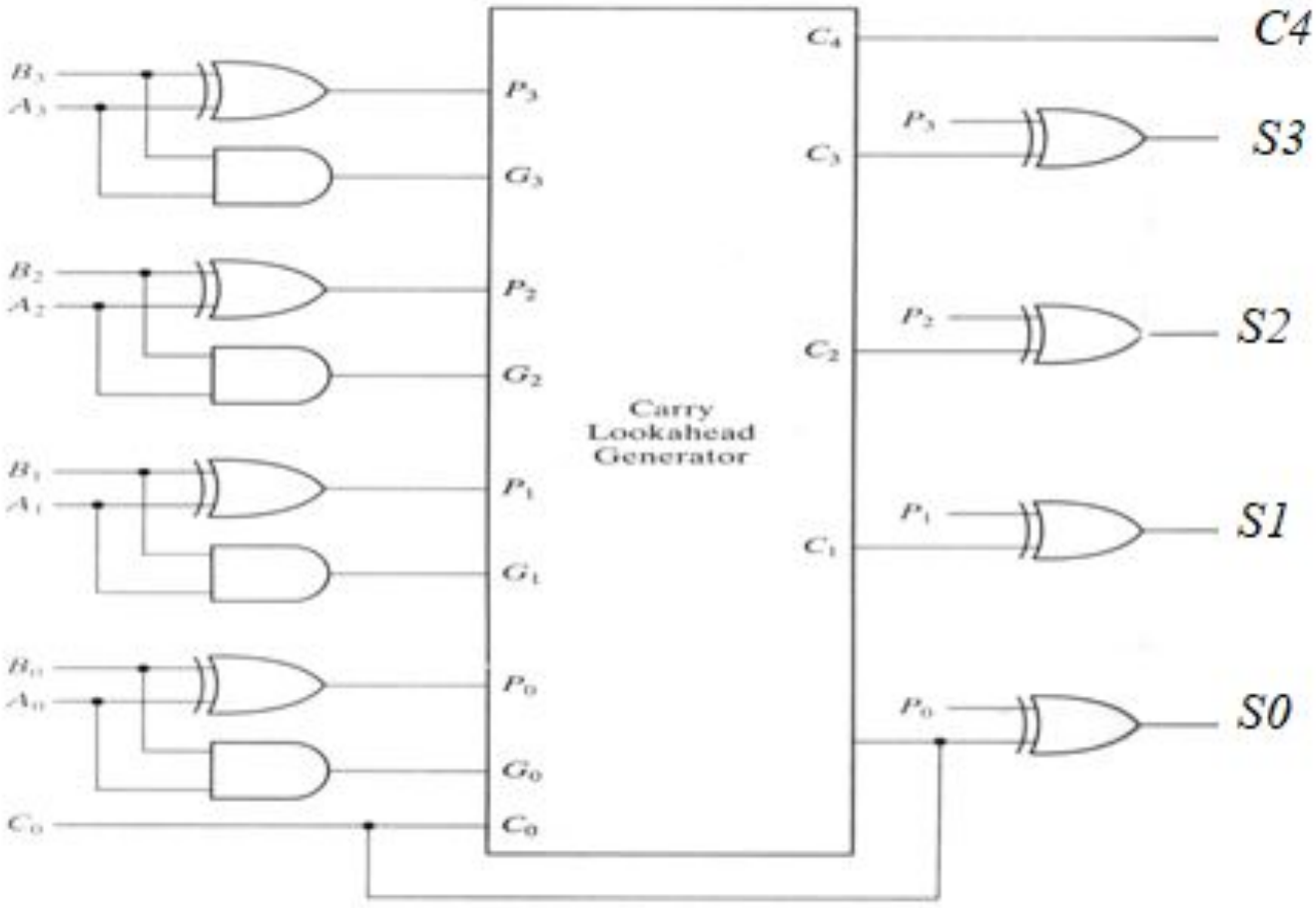
$$c_1 = G_0 + P_0 c_0$$

$$c_2 = G_1 + P_1 G_0 + P_1 P_0 c_0$$

$$c_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$$

$$c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$

4-bit Carry Look-ahead Adder



4-bit Carry Look-ahead Adder

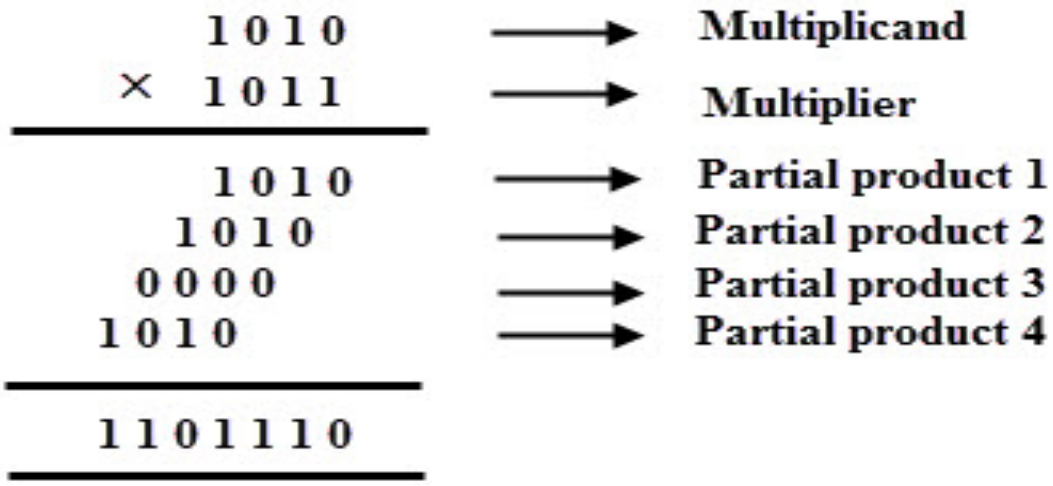
- independent of n,
- the n-bit addition process requires only four gate delays (against 2n)
- increasing ‘n’ increases gate fan-in requirements (C 4 – fan_in = 5)
- Longer version adders - cascading

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 multiplication of two numbers, say A and B (2 bits each),
- C = A x B.

Binary Multiplier

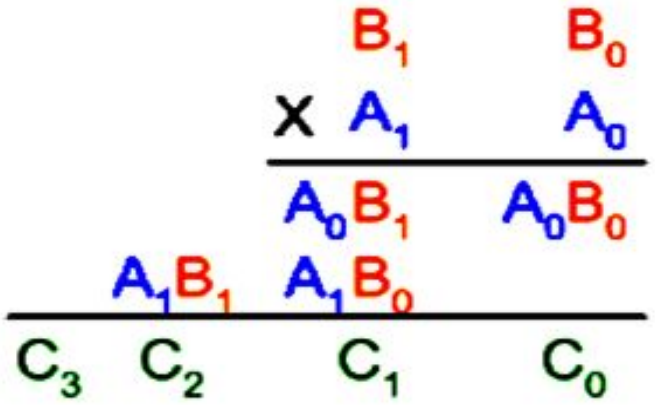
- Unsigned number multiplication
- two n-bit numbers; 2n-bit result

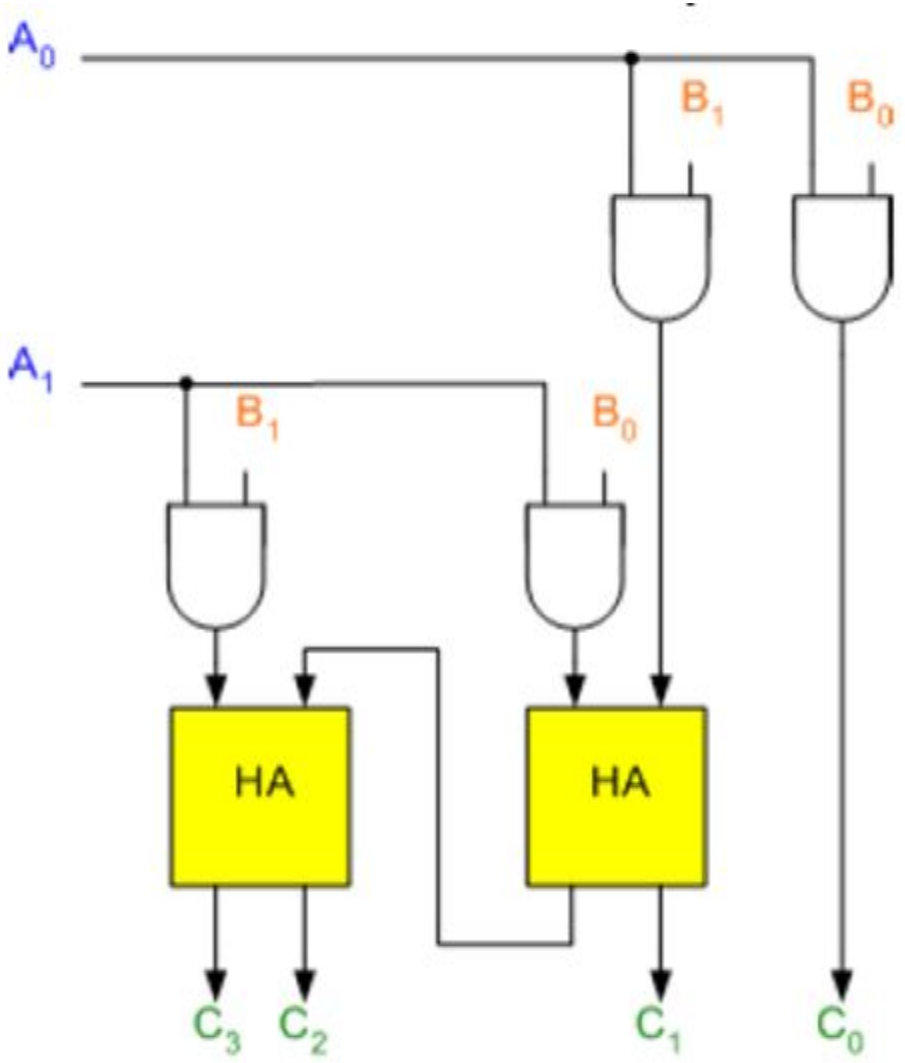


Binary Multiplier

- The first partial product is formed by multiplying the B1B0 by A0. The multiplication of two bits such as A0 and B0 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 B1B0 by A1 and is shifted one position to the left.
- 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.

Binary Multiplier





Binary Multiplier

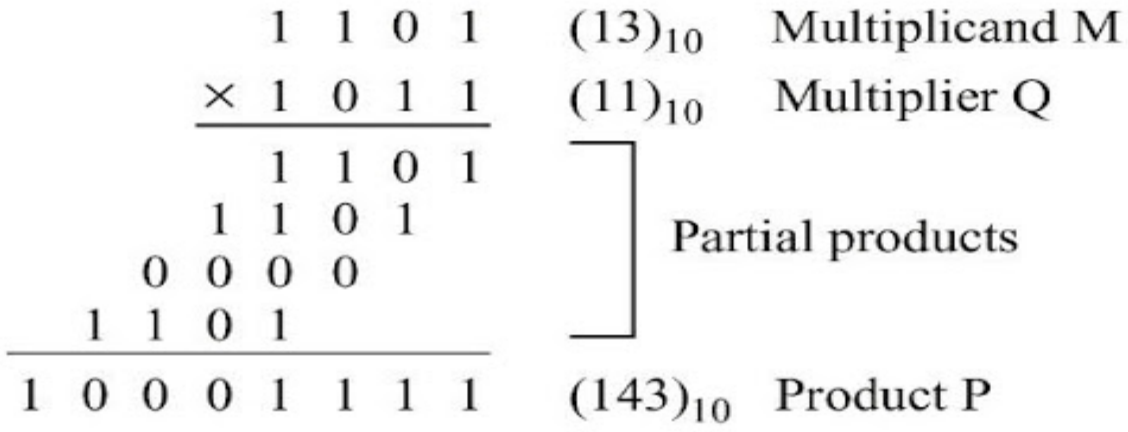
- The least significant bit of the product does not have to go through an adder, since it is formed by the output of the first AND gate as shown in the Figure.

Shift-and-Add Multiplier

- Shift-and-add multiplication is similar to the multiplication performed by paper and pencil.
- This method adds the multiplicand X to itself Y times, where Y denotes the multiplier.
- To multiply two numbers by paper and pencil, the algorithm is to take the digits of the multiplier one at a time from right to left, multiplying the multiplicand by a single digit of the multiplier and placing the intermediate product in the appropriate positions to the left of the earlier results.

Shift-and-Add Multiplier

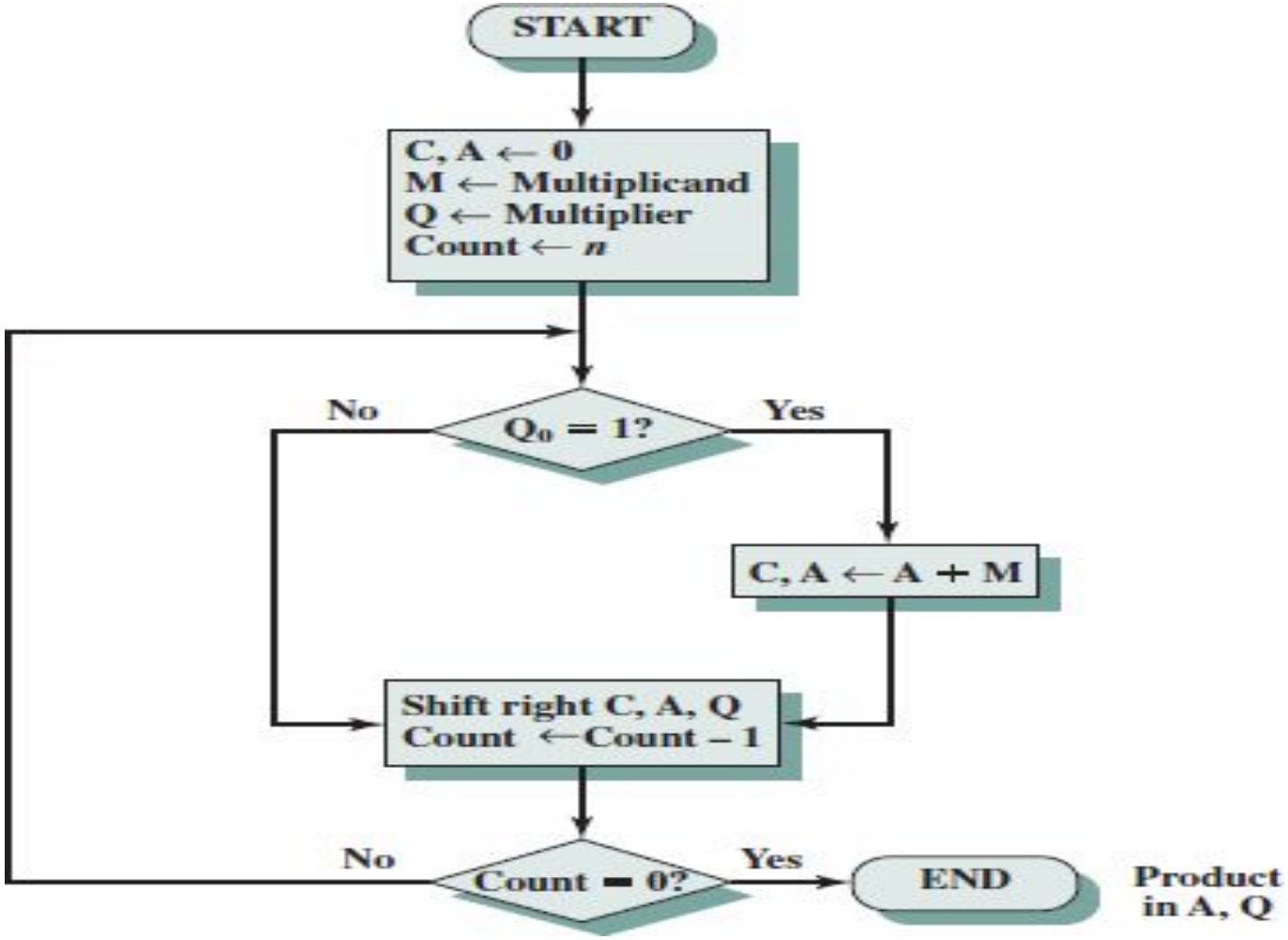
- As an example, consider the multiplication of two unsigned 4-bit numbers,
- 13 (1101) and 11 (1011).



Shift-and-Add Multiplier

- In the case of binary multiplication, since the digits are 0 and 1, each step of the multiplication is simple.
- If the multiplier digit is 1, a copy of the multiplicand (1 ×multiplicand) is placed in the proper positions;
- If the multiplier digit is 0, a number of 0 digits (0 × multiplicand) are placed in the proper positions.
- Consider the multiplication of positive numbers. The first version of the multiplier circuit, which implements the shift-and-add multiplication method for two n-bit numbers, is shown in Figure.

Shift - and - Add multiplier



Shift-and-Add Multiplier

- For Example, Perform the multiplication 13 x 11 (1101 x 1011). Finally, both A and Q contains the result of product.

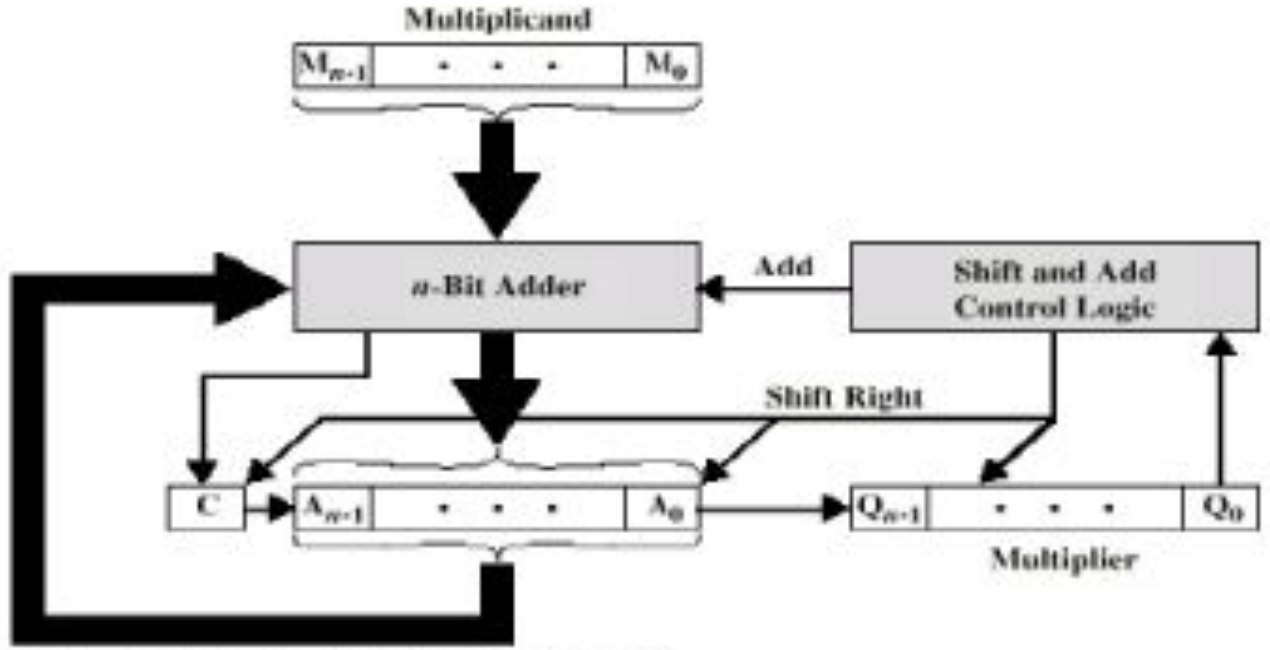
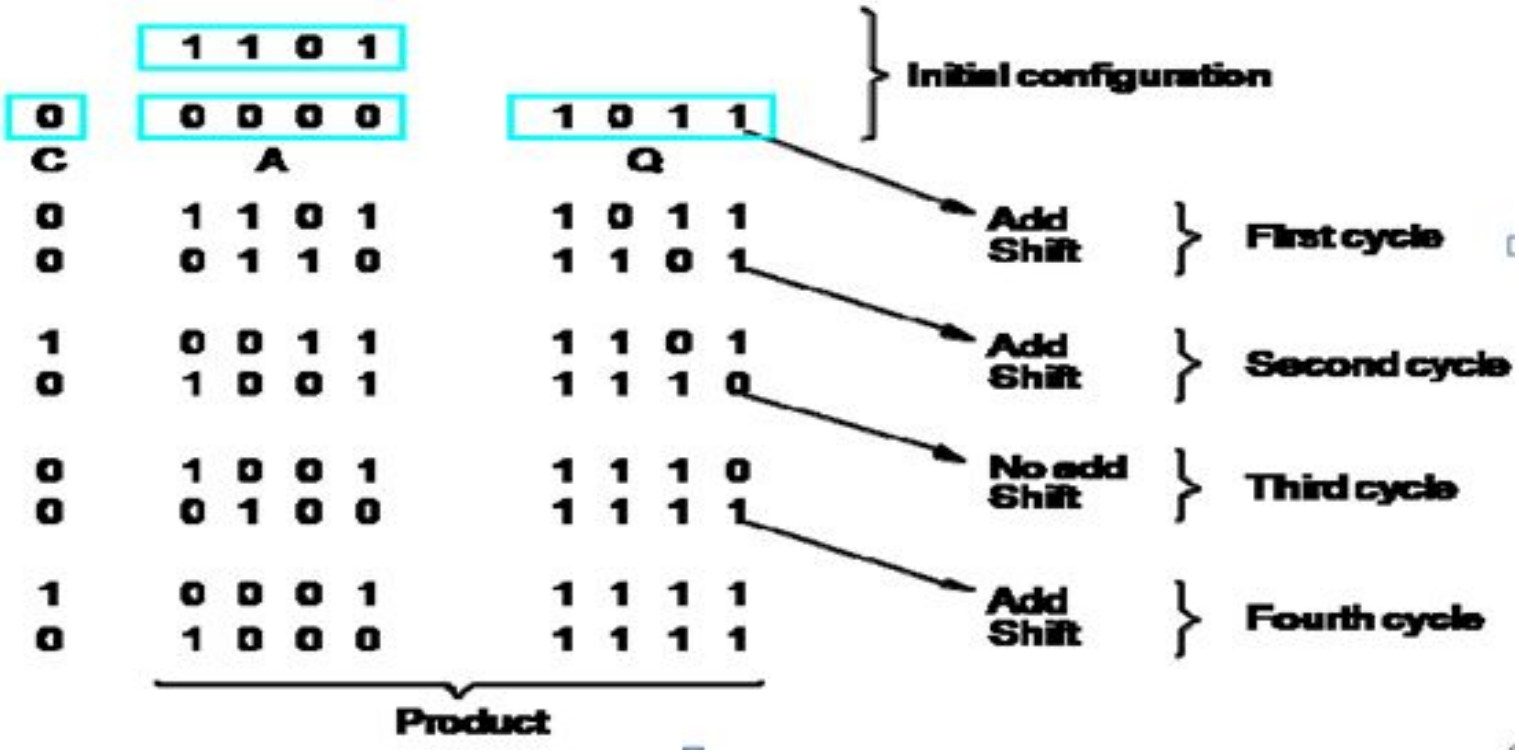


Fig: Block diagram of multiplication

Shift-and-Add Multiplier

Example 2:

- **A = 0000 M (Multiplicand) ? 13 (1 1 0 1) Q(Multiplier) ? 11 (1 0 1 1).**



11 × 13 $\quad$ 11 = 1011 (M) $\quad$ N = 4

13 = 1101 (Q)

N	C	A	Q	M	
4	0	0000	1101	1011	$q_0 = 1 \rightarrow$ A = A+M, RS, N-1
	0	1011	1101	1011	0000 + 1011 = 1011
		0101	1110	1011	4 − 1 = 3
3	0	0101	1110	1011	$q_0 = 0$, RS, N-1
		0010	1111	1011	3 − 1 = 2
2	0	0010	1111	1011	$q_0 = 1$, A = A+M, RS, N-1
	0	1101	1111	1011	0010 + 1011 = 1101
		0110	1111	1011	2 − 1 = 1
1	0	0110	1111	1011	$q_0 = 1$ A = A+M, RS, N-1
	1	0001	1111	1011	0110 + 1011 = 10001
	0	1000	1111	1011	N − 1 = 1 − 1 = 0

N = 0 Stop the process

Product = A Q

1000 1111

= 143

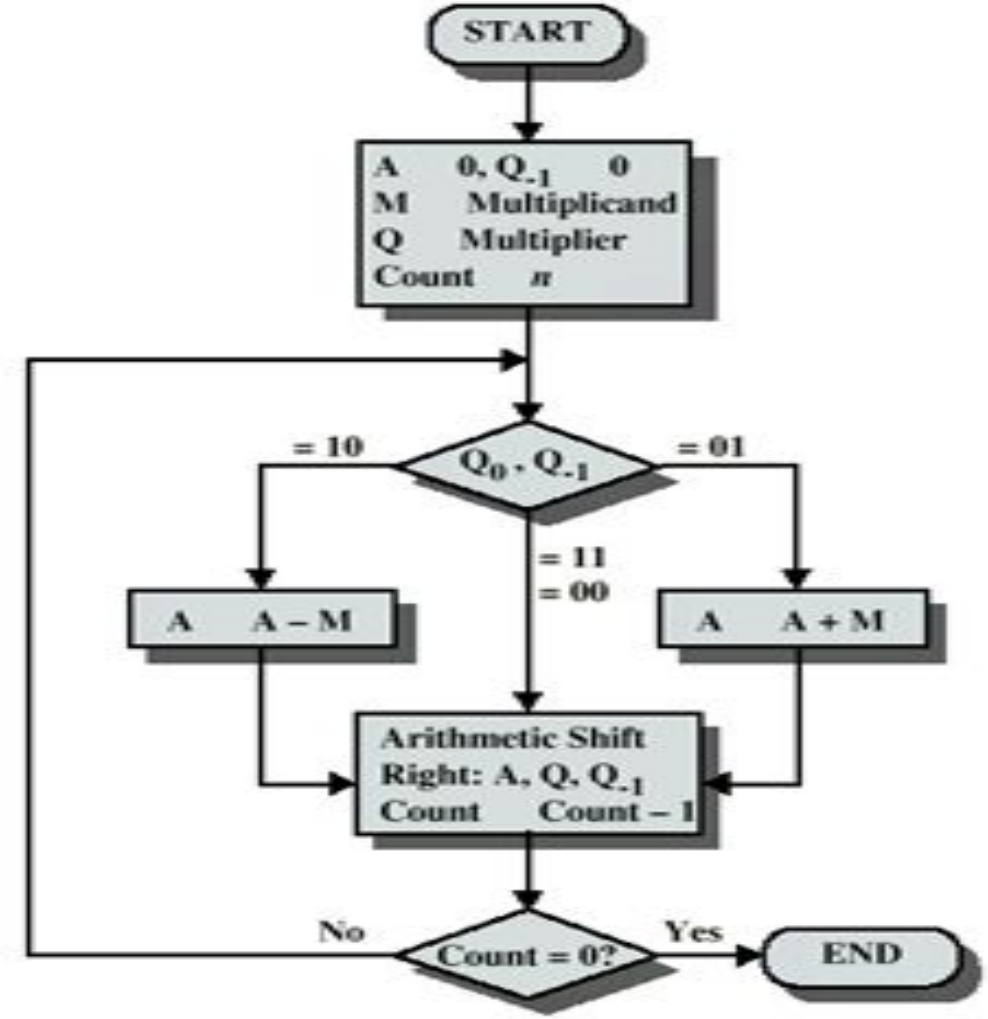
Try for 12 × 10

Signed Multiplication - Booth Algorithm

Signed Multiplication - Booth Algorithm

- A powerful algorithm for signed number multiplication is Booth's algorithm which generates a 2n bit product and treats both positive and negative numbers uniformly.
- This algorithm suggest that we can reduce the number of operations required for multiplication by representing multiplier as a difference between two numbers.

Signed Multiplication - Booth Algorithm



Signed Multiplication - Booth Algorithm

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

Signed Multiplication

BOOTH RECODING OF MULTIPLERS

Signed Multiplication

- Considering 2's-complement signed operands, what will happen to (-13)×(+11) if following the same method of unsigned multiplication?

					1	0	0	1	1	(- 13)
					0	1	0	1	1	(+11)
1	1	1	1	1	1	0	0	1	1	
1	1	1	1	1	0	0	1	1		
0	0	0	0	0	0	0	0			
1	1	1	0	0	1	1				
0	0	0	0	0	0					
1	1	0	1	1	1	0	0	0	1	(- 143)

Sign extension is shown in blue

Sign extension of negative multiplicand.

Example for Signed Number Multiplication

① -7×3 $\quad q = 3 = 0011$ $\quad q_0 = 0$

$M = -7$ $\qquad N = 4$

$7 = 0111 \xrightarrow{1's} 1000 \xrightarrow{2's} 1001 \qquad -M = 0111$

N	A	q	q_0	M	Steps	①	②	3
4	0000	0011	0	1001	$q_0 = 0$ $q_1 = 1$	A = A − M = A + (−M) = 0000 0111 0111	ARS	N = N − 1 4 − 1 = 3
	0111 0111	0011 1111	0	1001				
3	0011	1001	1	1001				
	0001	1100	1	1001	$q_0 = 1$ $q_1 = 1$		ARS	N = N − 1
2	0001	1100	1	1001	$q_0 = 1$ $q_1 = 0$	A = A + M 0001 1001 1010	ARS	N = N − 1
	1010 1101	1100 0110	1 0	1001 1001				
1	1101	0110	0	1001	$q_0 = 0$ $q_1 = 0$		ARS	N = N − 1
	1110	1011	0	1001				
0	1110	1011	0	1001				

Product = AQ

= 1110 1011

= − (1110 1011) $\quad$ (1's)

= 0001 0100 $\quad$ (2's, +1)

0001 0101 $\quad$ ($2^4 2^3 2^2 2^1 2^0$)

= 21

∴ −21

Example 2

Perform Multiplication for $(-15) \times (-13)$

$M = -15$

$15 =$ S 8 4 2 1
0 1 1 1 1
1's = 1 0 0 0 0
+1
2's = 1 0 0 0 1

$M = 1\ 0\ 0\ 0\ 1$

$-M = 0\ 1\ 1\ 1\ 1$

$Q = -13$

$13 =$ S 8 4 2 1
0 1 1 0 1
1's → 1 0 0 1 0
+1
2's → 1 0 0 1 1

$Q = 10011$, $q_0 = 0$

$N = 5$

N	A	q	q_0	M	$q_0=0$ A=A-M $q_1=1$ A=A+(-M)
5	00000	10011	0	10001	
	01111	10011	0	10001	00000 + 01111 = 01111 → ARS → N=N-1
	00111	11001	1	10000	
4	00111	11001	1	10000	$q_0=1, q_1=1 \Rightarrow$ ARS → N-1
	00011	11100	1	10000	
3	00011	11100	1	10001	$q_0=1, q_1=1$ = A=A+M
	10100	11100	1	10001	00011 + 10001 = 10100 → ARS → N-1
	11010	01110	0	10001	
2					

N	A	q	q_0	M	$q_0=0$ $q_1=0$ → ARS → N-1
2	11010	01110	0	10001	
	11101	00111	0	10001	
1	11101	00111	0	10001	$q_0=0$ A=A-M, $q_1=1$, A=A+(-M)
	01100	00111	0	10001	11101 + 01111 = 01100 → ARS → N-1
0	00110	00011	1	10001	

Product = AQ

00110 00011 = 195

$-15 \times -13 = 195$

Try for 1) 15×-13

2) -6×7

The BOOTH RECODED MULTIPLIER

- Booth multiplication reduces the number of additions for intermediate results, but can sometimes make it worse as we will see.
- Booth multiplier recoding table

Multiplier		Version of multiplicand selected by bit
Bit *i*	Bit *i*-1	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

The BOOTH RECODED MULTIPLIER

- Booth Recoding:

(i) 30_{10} : 0 1 1 1 1 0 0

- +1 0 0 0 -1 0
- (ii) 100_{10}: 0 1 1 0 0 1 0 0 0
-
- +1 0 -1 0 +1 -1 0 0
- (iii) 985_{10}: 0 0 1 1 1 1 0 1 1 0 0 1 0
- 0 +1 0 0 0 -1 +1 0 -1 0 +1 -1

BOOTH Algorithm

Booth algorithm treats both +ve &
-ve operands equally.
(a) + Md X + Mr

Ex: 0 1 1 0 1 (+13) X 0 1 0 1 1 (+11)

```
        +1-1+1 0 -1
-------------------------
1 1 1 1 1 1 0 0 1 1
0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 0 1
1 1 1 0 0 1 1
0 0 1 1 0 1

-------------------------
0 0 1 0 0 0 1  1 1 1 (+143)
-------------------------
```

BOOTH Algorithm

Booth algorithm treats both +ve &
-ve operands equally.
(b) - Md X + Mr

Ex: 1 0 0 1 1 (-13) X 0 1 0 1 1 (+11)

+1-1+1 0 -1

---

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

---

1 0 1 0 0 0 0 0 0 1 (-143)

BOOTH Algorithm

Booth algorithm treats both +ve &
-ve operands equally.
(c) + Md X - Mr

Ex: 0 1 1 0 1 (+13) X 1 0 1 0 1 (-11)

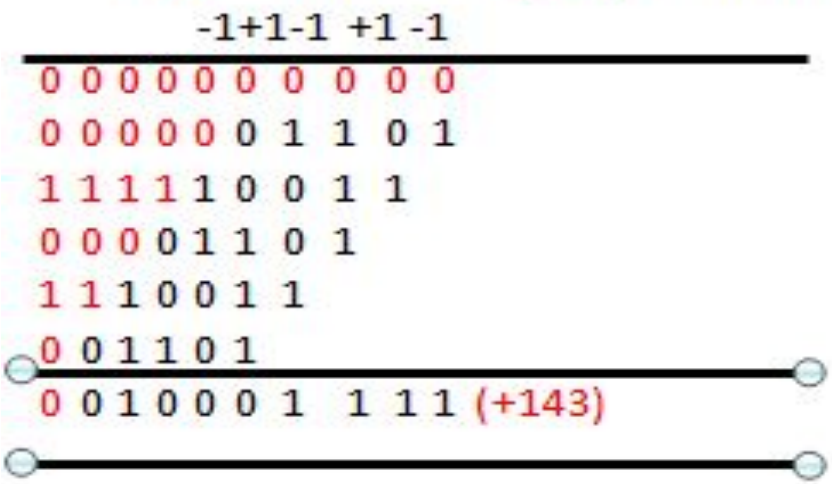
-1+1-1 +1-1

0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 0 0 1 1
0 0 0 0 0 1 1 0 1
1 1 1 1 0 0 1 1
0 0 0 1 1 0 1
1 1 0 0 1 1
1 1 0 1 1 1 0 0 0 1 1 0 1 0 0 0 1 1 1 1 (-143)

BOOTH Algorithm

Booth algorithm treats both +ve &
-ve operands equally.
(d) - Md X - Mr

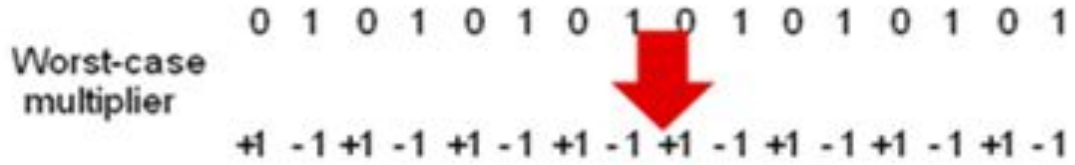
Ex: 1 0 0 1 1 (-13) X 0 0 1 0 1 (-11)



BOOTH Algorithm

Count = 4 Vs 8 speed improvement

Count = 7 Vs 9 no speed improvement



Count = 16 Vs 8 speed worsened.

On an average no improvement in speed

Example 2

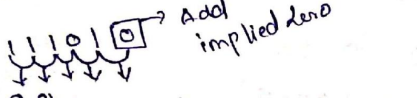
-12×-3

Multiplicand $12 = 01100 \xrightarrow{1's} 10011 \xrightarrow[+1]{2's} 10100$

Multiplier $3 = 00011 \longrightarrow 11100 \xrightarrow[+1]{} 11101$

$-12 = 10100$, $-3 = 11101$

Recode the Multiplier (-3)



∴ Recoded value is $0\ 0\ -1\ +1\ -1$

Sign bit extension

$10100 \times 0\ 0\ -1\ +1\ -1$

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

1) 0 0 0 1 0 0 1 0 0

$2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

$32 + 4 = 36$

When doing Multiplication by -1, the result is taken 2's Complement of Multiplicand

2's of 12 is 01100

$-12 \times -3 = 36$

Try for -11×6

FAST MULTIPLICATION

1. BIT PAIR RECODING OF MULTIPLIERS
2. CARRY SAVE ADDITION OF SUMMANDS

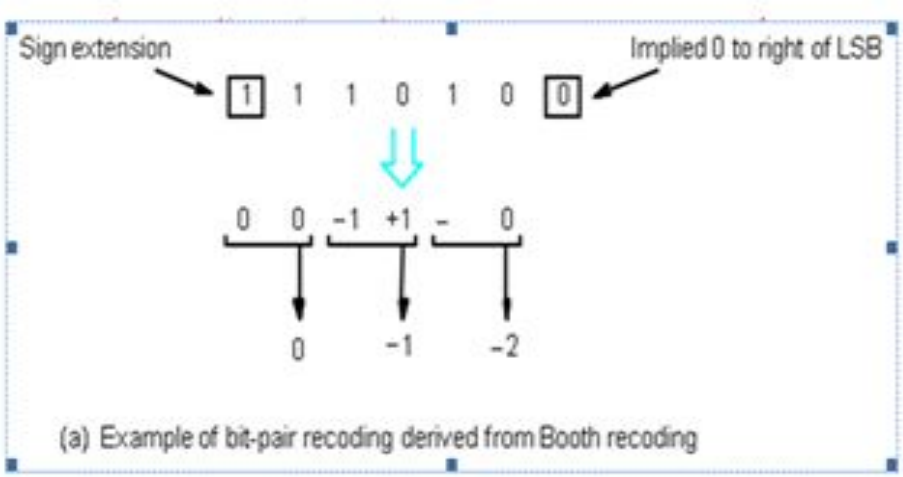
Fast Multiplication

- There are two techniques for speeding up the multiplication operation.
- The first technique guarantees that the maximum number of summands (versions of the multiplicand) that must be added is *n/2* for n-bit operands.
- The second technique reduces the time needed to add the summands (carry-save addition of summands method).

Bit-Pair Recoding of Multipliers

- This bit-pair recoding technique halves the maximum number of summands. It is derived from the Booth algorithm.
- Group the Booth-recoded multiplier bits in pairs, and observe the following: The pair (+1 -1) is equivalent to the pair (0 +1).
- That is, instead of adding —1 times the multiplicand M at shift position i to + 1 x M at position i + 1, the same result is obtained by adding +1 x M at position I Other examples are: (+1 0) is equivalent to (0 +2),(-l +1) is equivalent to (0 —1). and so on

Bit-Pair Recoding of Multipliers



original bit pair i+1	i	Bit to right i-1	Bit pair Recoded Yi+1	vi	Multiplier value
0	0	0	0	0	0
0	0	1	0	1	+1
0	1	0	1	-1	+1
0	1	1	1	0	+2
1	0	0	-1	0	-2
1	0	1	-1	1	-1
1	1	0	0	-1	-1
1	1	1	0	0	0

BIT PAIR RECODING OF MULTIPLIER

i+1	i	i-1	Multiplicand Selected position i
0	0	0	+0 x M
0	0	1	+1 x M
0	1	0	+1 x M
0	1	1	+2 x M
1	0	0	-2 x M
1	0	1	-1 x M
1	1	0	-1 x M
1	1	1	0 x M

Bit-Pair Recoding of Multipliers

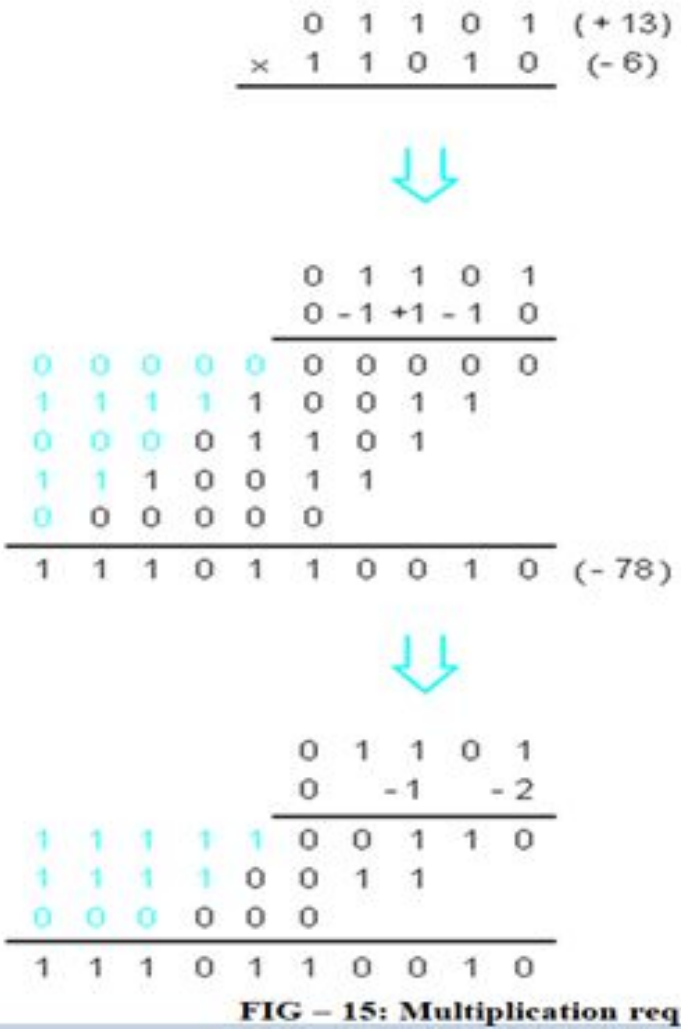


FIG – 15: Multiplication requiring n/2 summands.

Carry-Save Addition of Summands

- A **carry-save adder** is a type of digital adder, used to efficiently compute the sum of three or more binary numbers.
- A carry-save adder (CSA), or 3-2 adder, is a very fast and cheap adder that does not propagate carry bits.
- A Carry Save Adder is generally used in binary multiplier, since a binary multiplier involves addition of more than two binary numbers after multiplication.
- It can be used to speed up addition of the several summands required in multiplication
- It differs from other digital adders in that it outputs two (or more) numbers, and the answer of the original summation can be achieved by adding these outputs together.
- A big adder implemented using this technique will usually be much faster than conventional addition of those numbers.

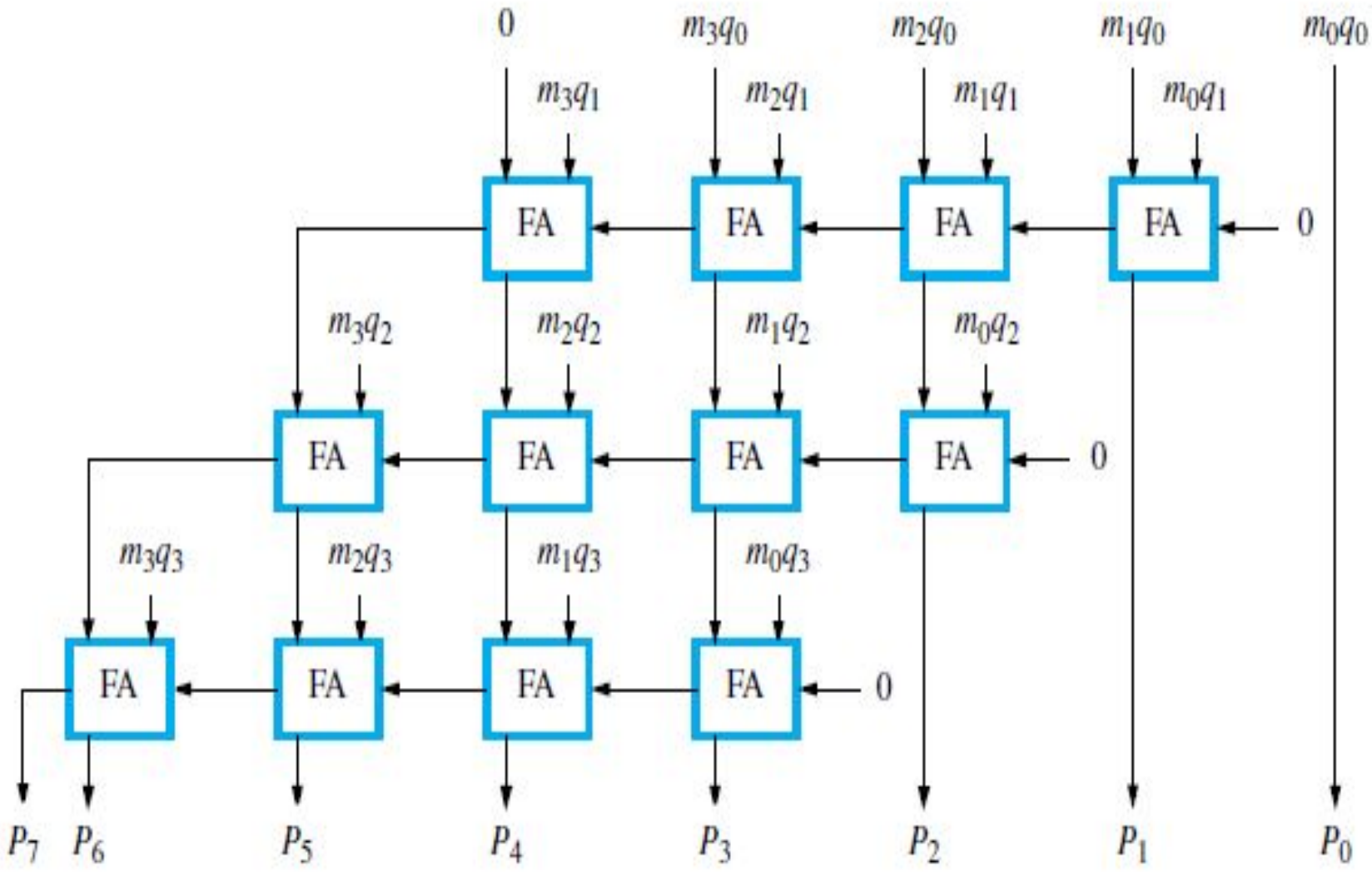
Fast Multiplication

- Bit pair recoding reduces summands by a factor of 2
- Summands are reduced by carry save addition
- Final product can be generated by using carry look ahead adder

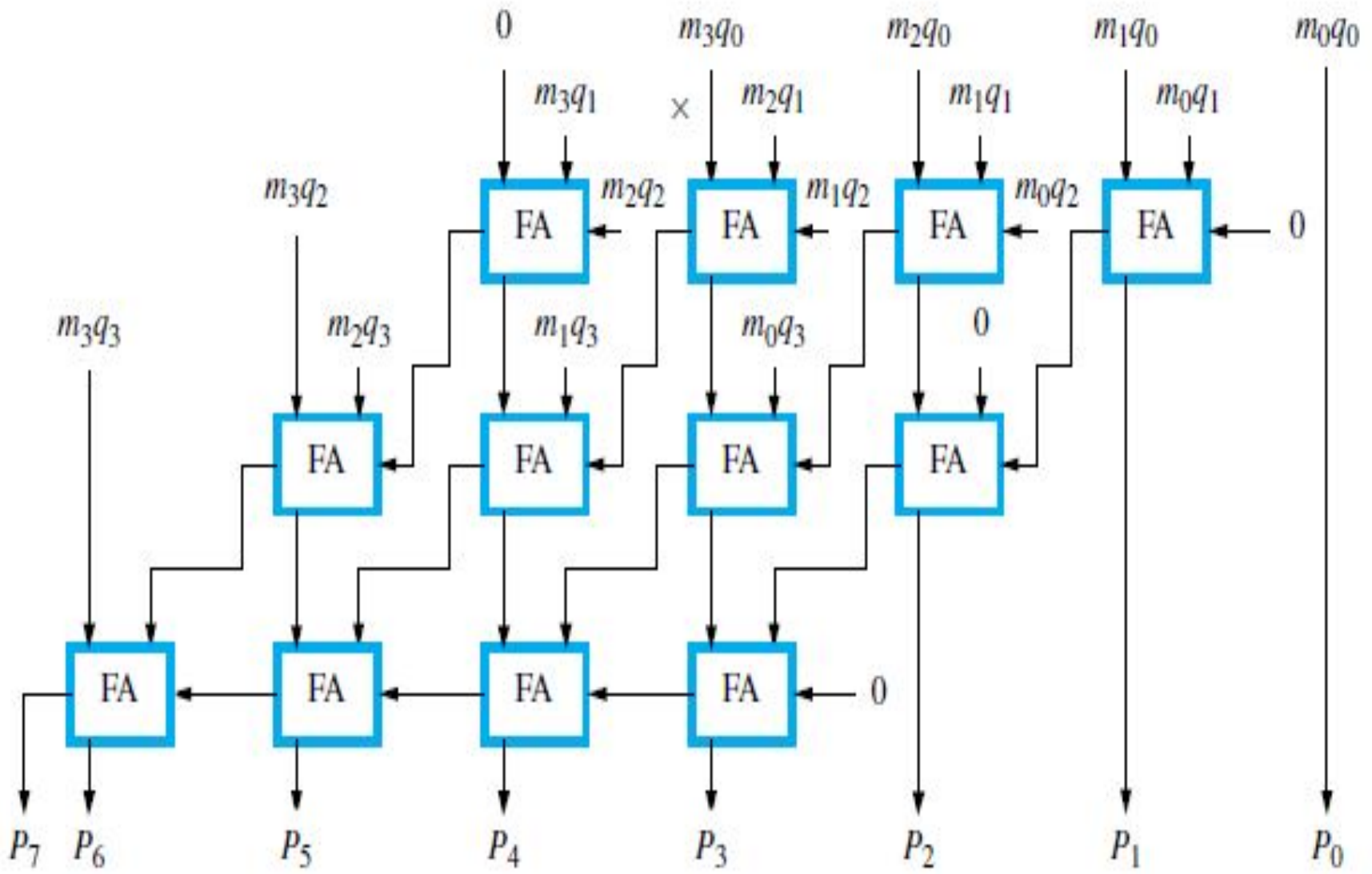
Carry-Save Addition of Summands

- **Disadvantage** of the **Ripple Carry Adder -** Each full adder has to wait for its carry-in from its previous stage full adder. This increase propagation time. This causes a delay and makes ripple carry adder extremely slow. RCAr is very slow when adding many bits.
- **Advantage of the Carry Look ahead Adder -** This is an improved version of the Ripple Carry Adder. Fast parallel adder. It generates the carry-in of each full adder simultaneously without causing any delay. So, CLAr is faster (because of reduced propagation delay) than RCAr.
- **Disadvantage the Carry Look-ahead Adder** - It is costlier as it reduces the propagation delay by more complex hardware. It gets more complicated as the number of bits increases.

Ripple Carry Array



Carry Save Array

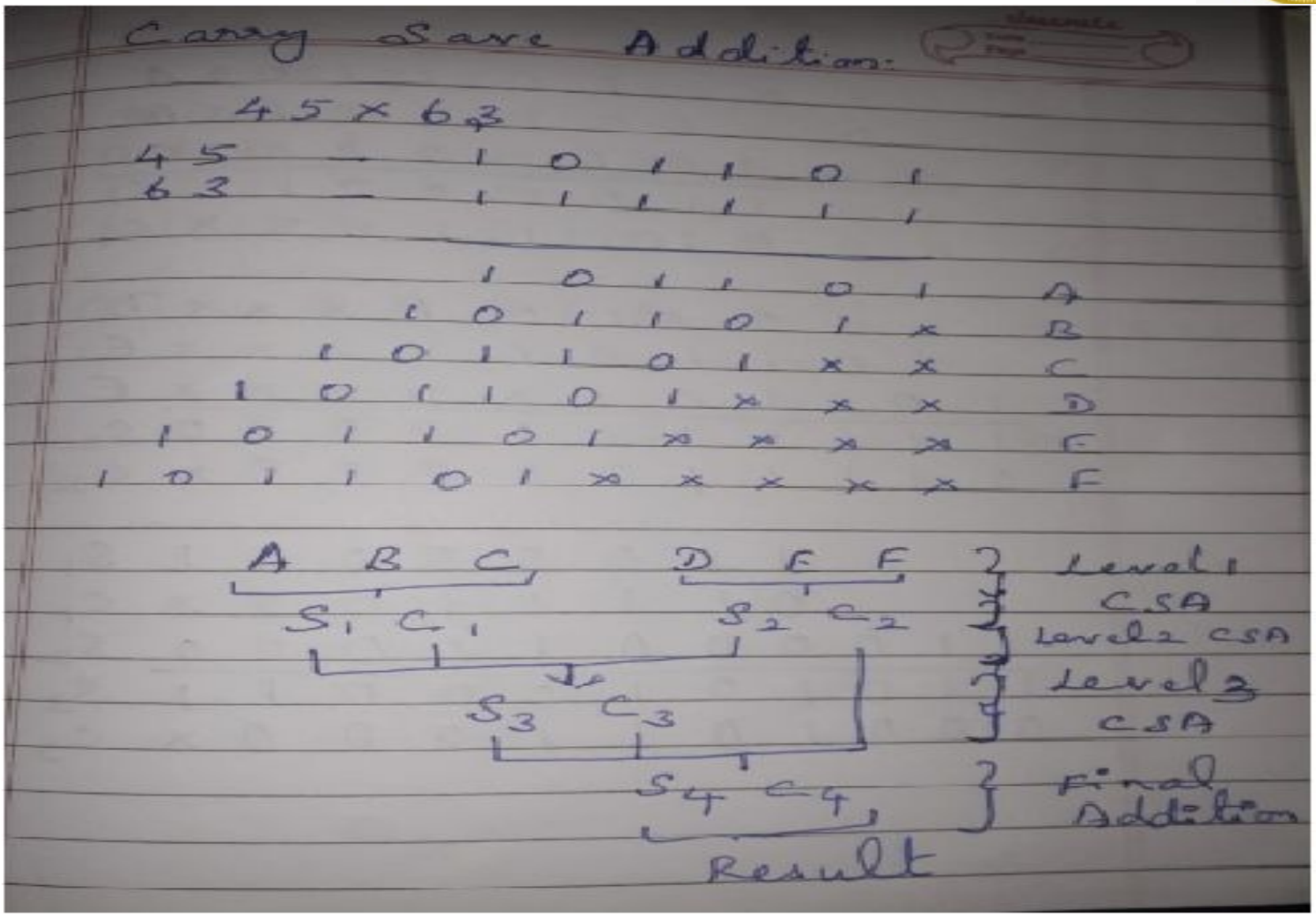


Carry-Save Addition of Summands

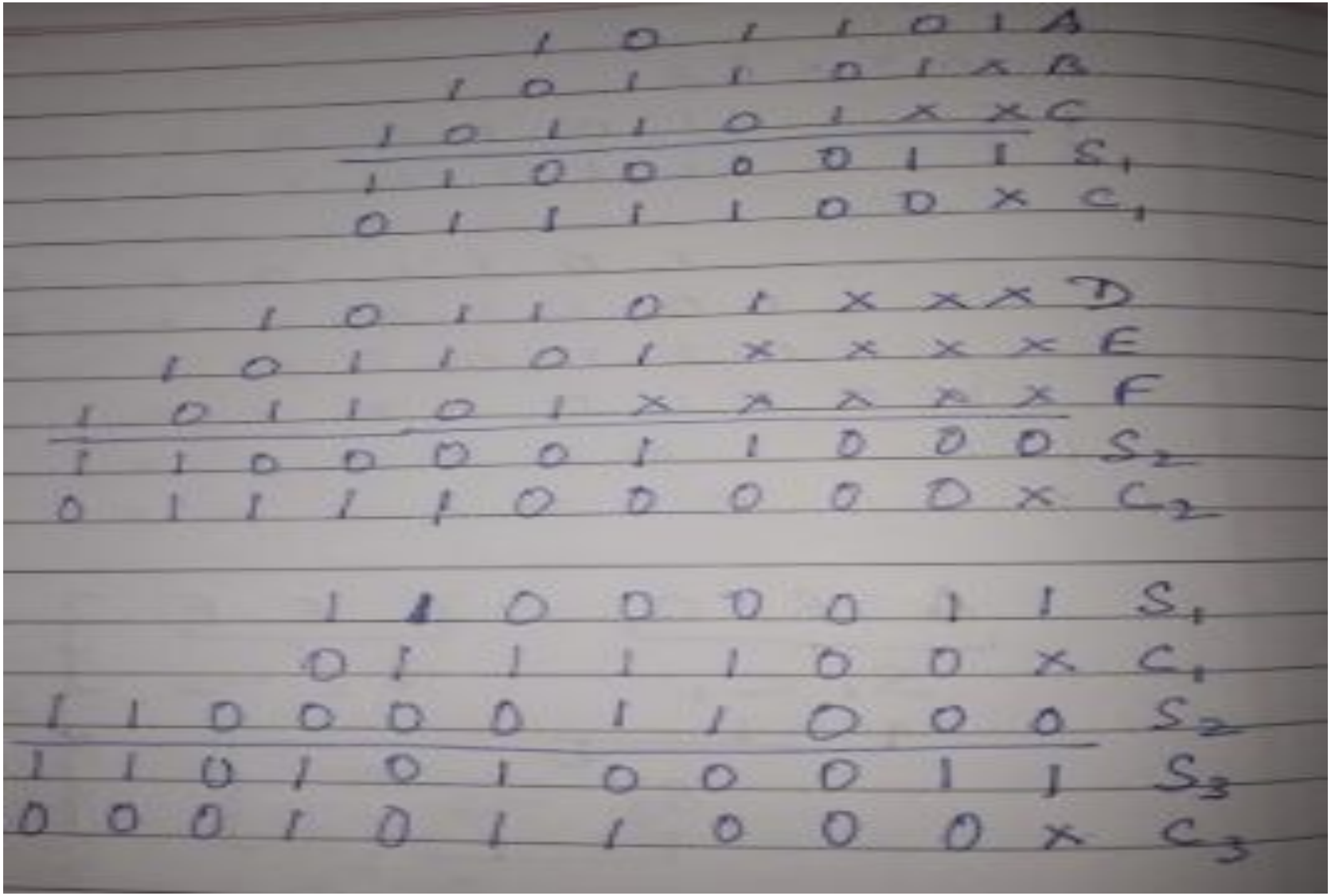
- Consider the addition of many summands, We can:

- Group the summands in threes and perform carry-save addition on each of these groups in parallel to generate a set of S and C vectors in one full-adder delay
- Group all of the S and C vectors into threes, and perform carry-save addition of them, generating a further set of S and C vectors in one more full-adder delay
- Continue with this process until there are only two vectors remaining
- They can be added in a Ripple Carry Adder (RPA) or Carry Look-ahead Adder (CLA) to produce the desired product

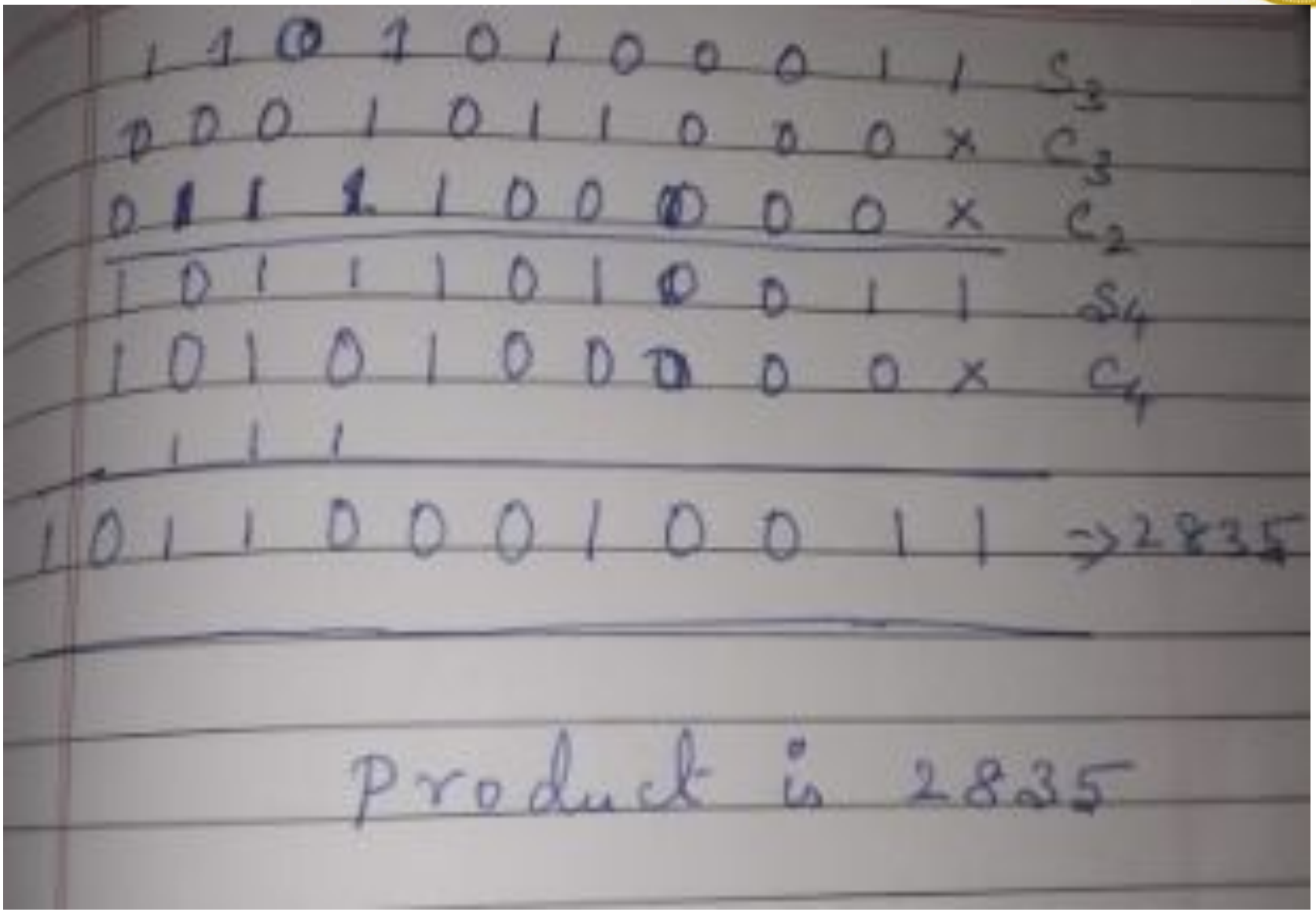
Carry-Save Addition of Summands



Carry-Save Addition of Summands

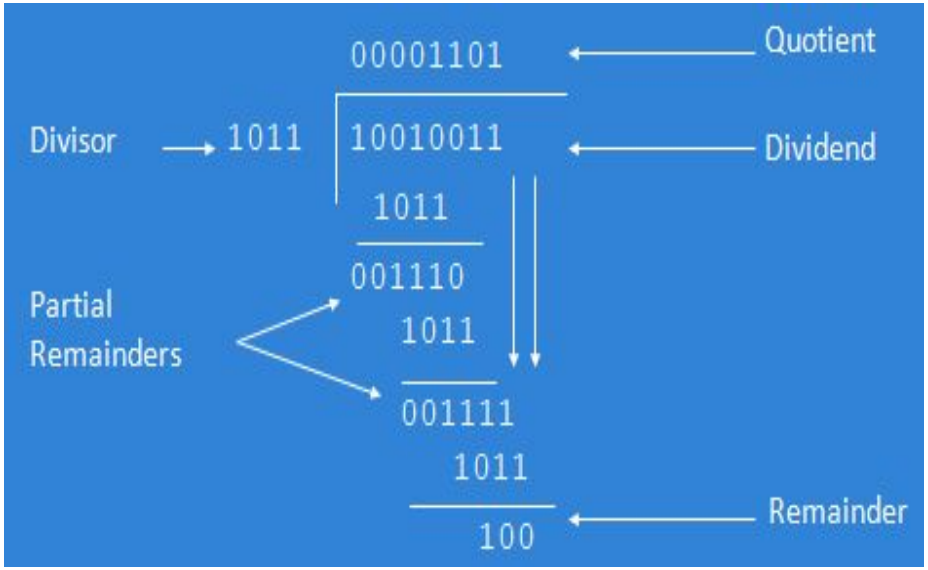


Carry-Save Addition of Summands



Integer Division

- More complex than multiplication
- Negative numbers are really bad!
- Based on long division



Integer Division

- Decimal Division
- Binary Division

```
      21
13 ) 274
     26
     ---
      14
      13
     ---
       1
```

```
          10101
      ------------
1101 ) 100010010
       1101
       ----
        10000
         1101
         ----
          1110
          1101
          ----
             1
```

Figure: Longhand division examples

Integer Division

Longhand Division operates as follows:

- Position the divisor appropriately with respect to the dividend and performs a subtraction.
- If the remainder is zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and another subtraction is performed.
- If the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor is repositioned for another subtraction

Restoring Division

- Similar to multiplication circuit
- An n-bit positive divisor is loaded into register M and an n-bit positive dividend is loaded into register Q at the start of the operation.
- Register A is set to 0
- After the division operation is complete, the n-bit quotient is in register Q and the remainder is in register A.
- The required subtractions are facilitated by using 2's complement arithmetic.
- The extra bit position at the left end of both A and M accommodates the sign bit during subtractions.

Restoring Division

Strategy for unsigned division:

Shift the dividend one bit at a time starting from MSB into a register.
Subtract the divisor from this register.
If the result is negative ("didn't go"):
- Add the divisor back into the register.
- Record 0 into the result register.

If the result is positive:
- Do not restore the intermediate result.
- Set a 1 into the result register.

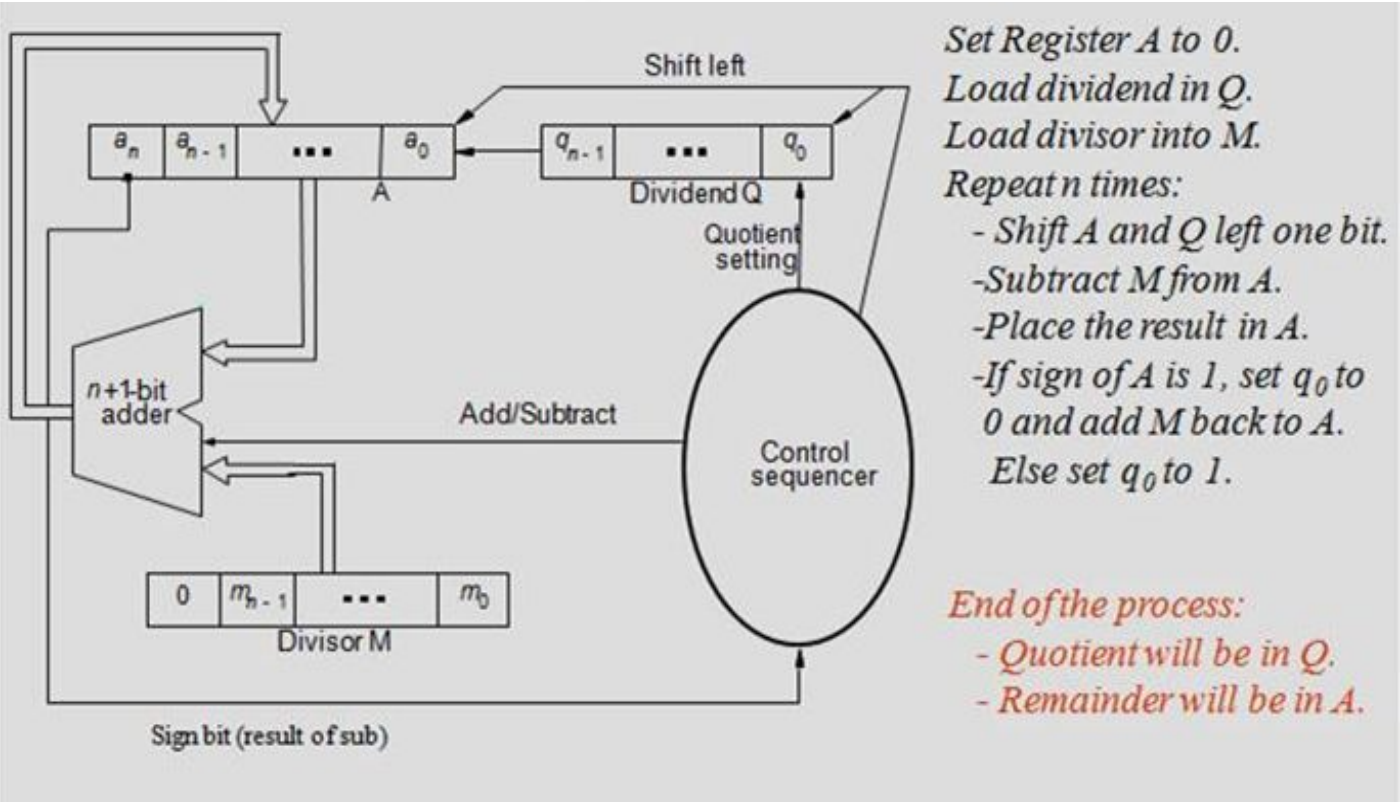


Figure: Logic Circuit arrangement for binary Division (Restoring)

Restoring Division

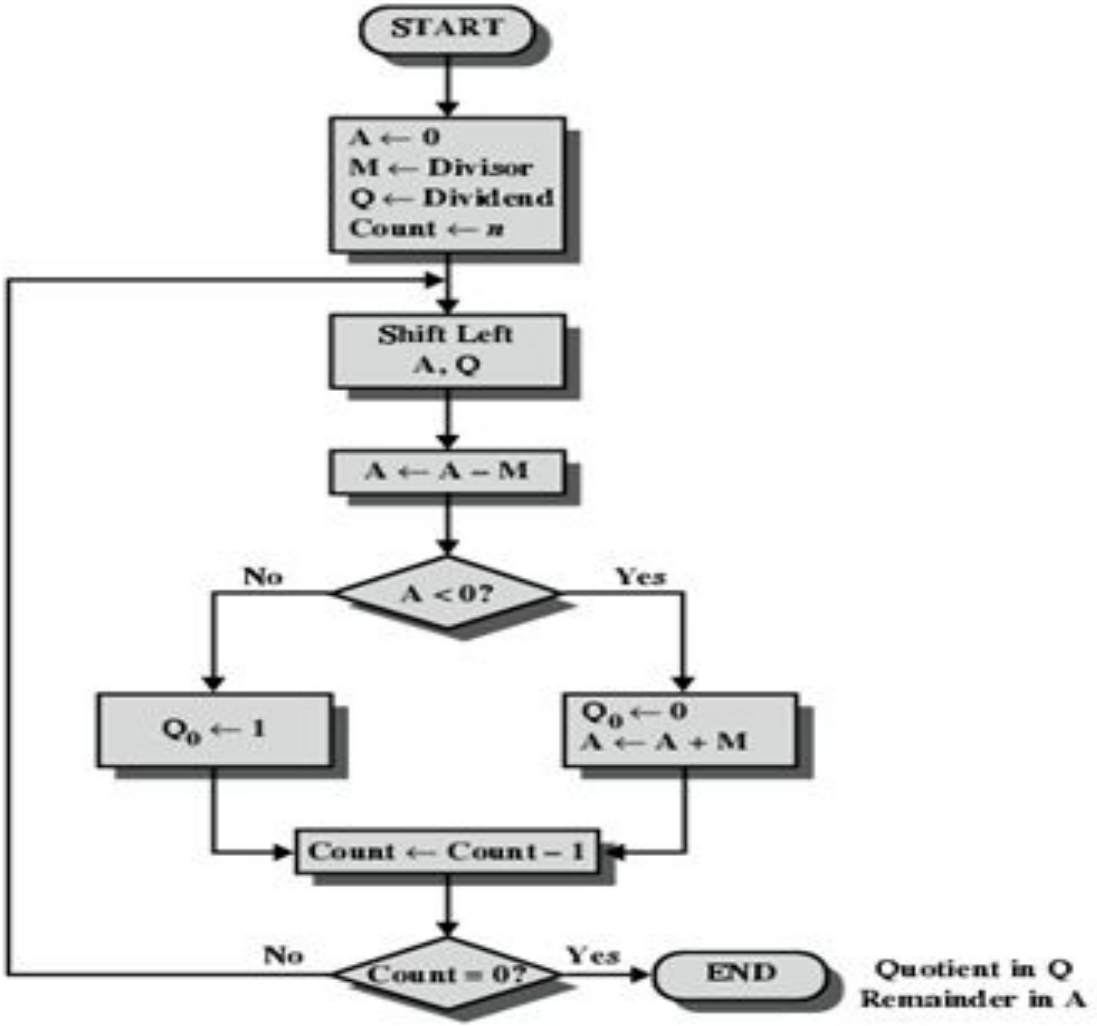
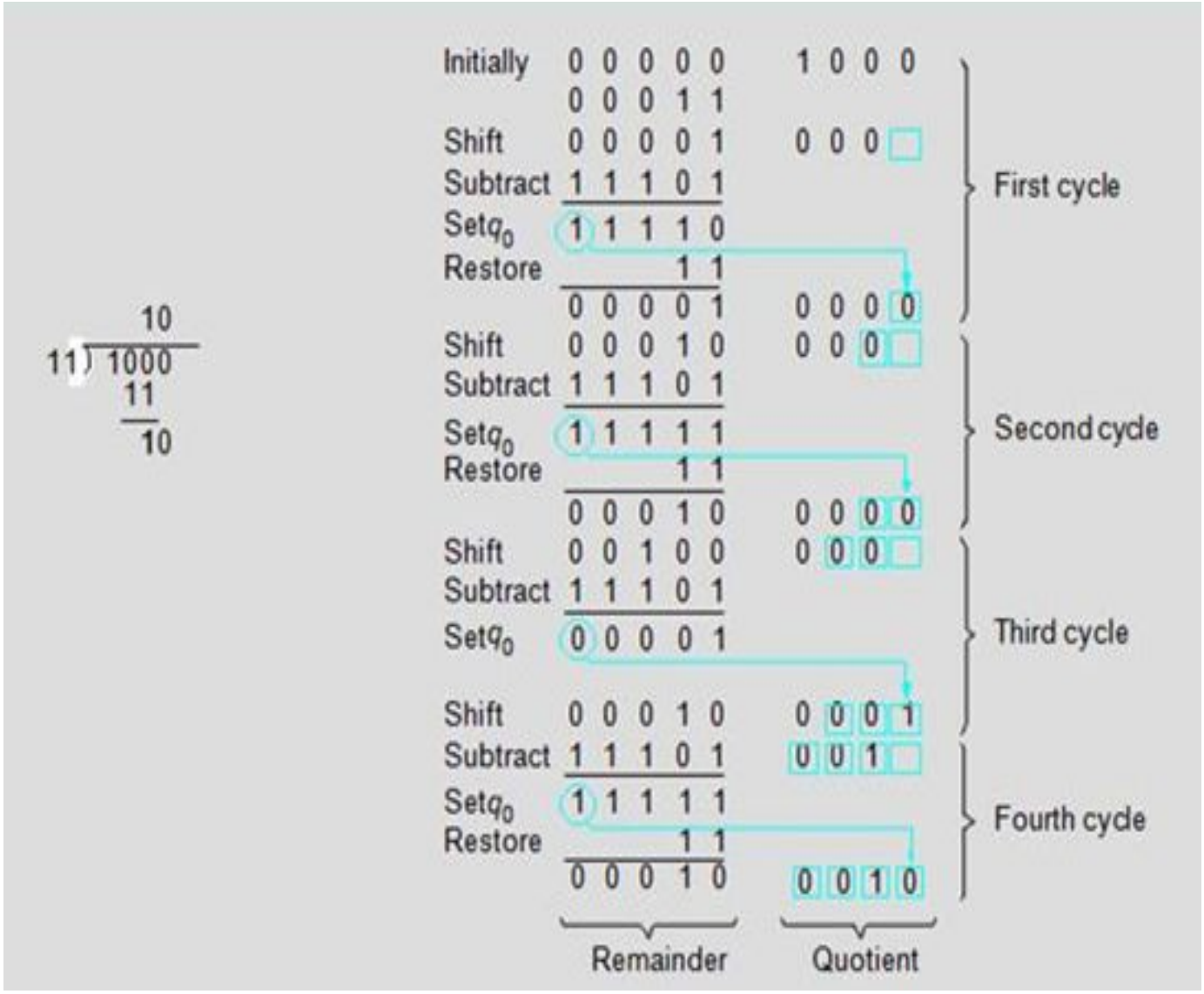
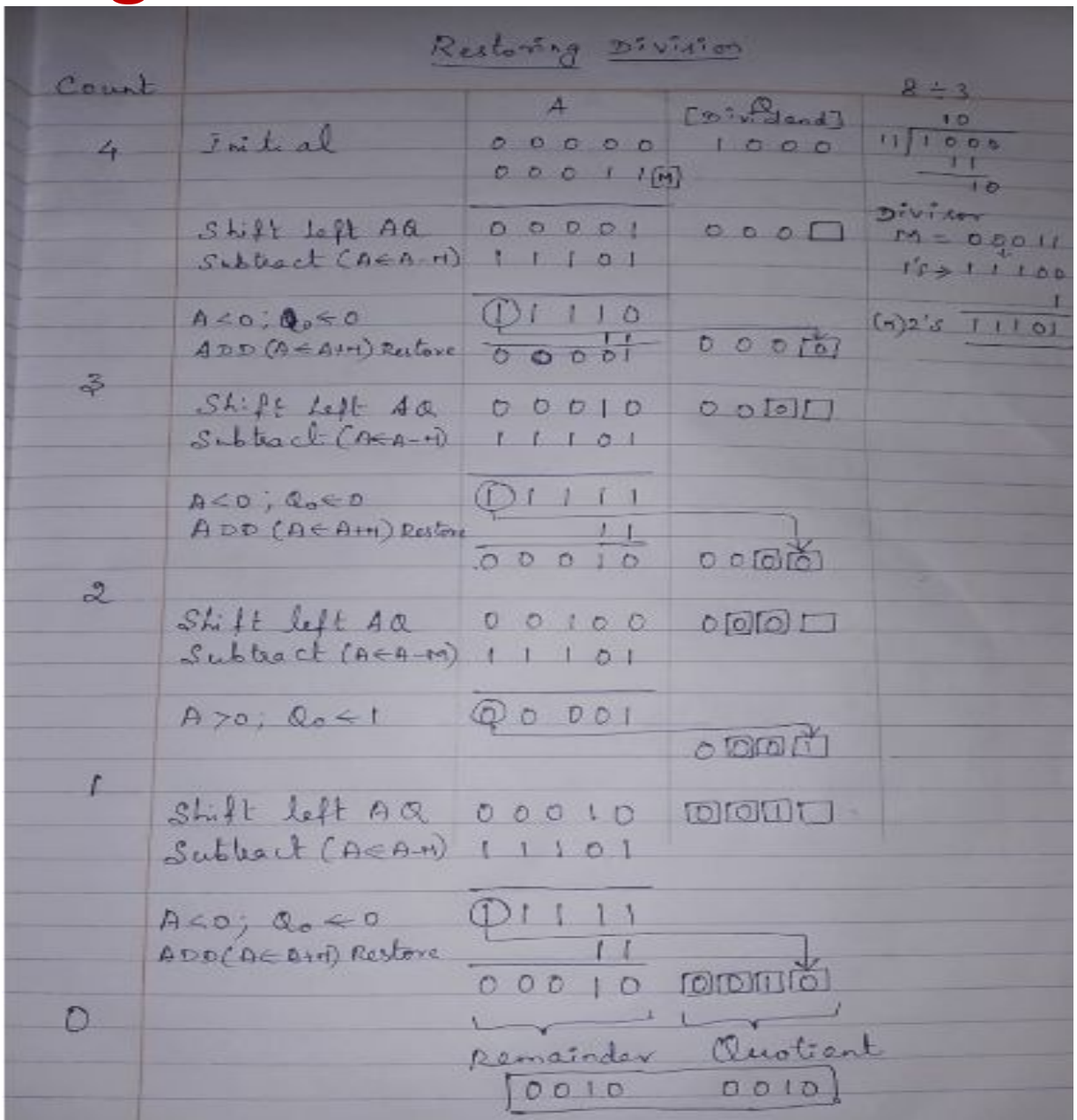


Figure: Flowchart for Restoring Division

Restoring Division



Restoring Division



Non-Restoring Division

- Initially Dividend is loaded into register Q,

  and n-bit Divisor is loaded into register M
- Let M' is 2's complement of M
- Set Register A to 0
- Set count to n
- SHL AQ denotes shift left AQ by one position leaving Q_0 blank.
- Similarly, a square symbol in Q_0 position denote, it is to be calculated later

Non-Restoring Division

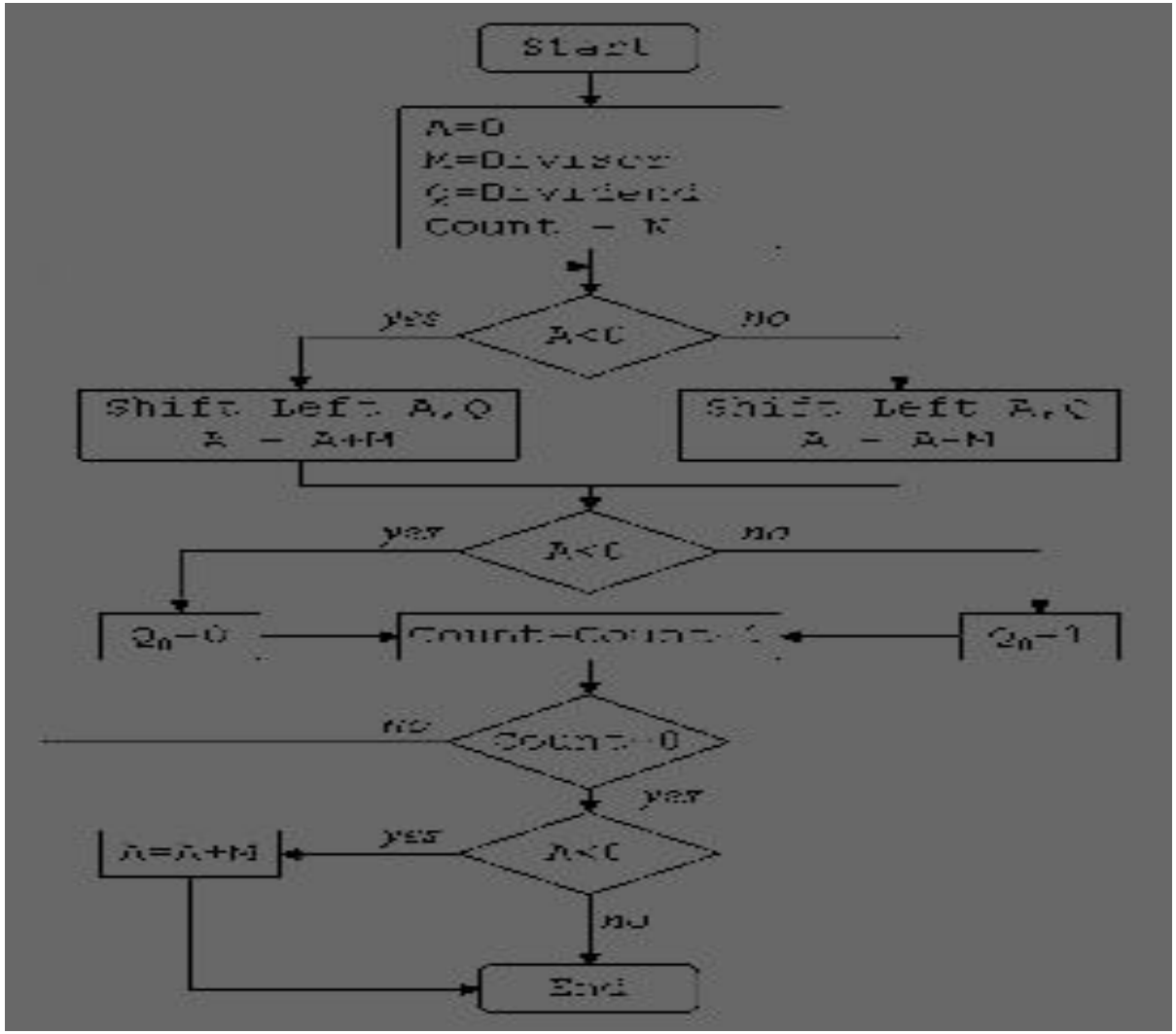


Figure: Flowchart for Non-Restoring Division

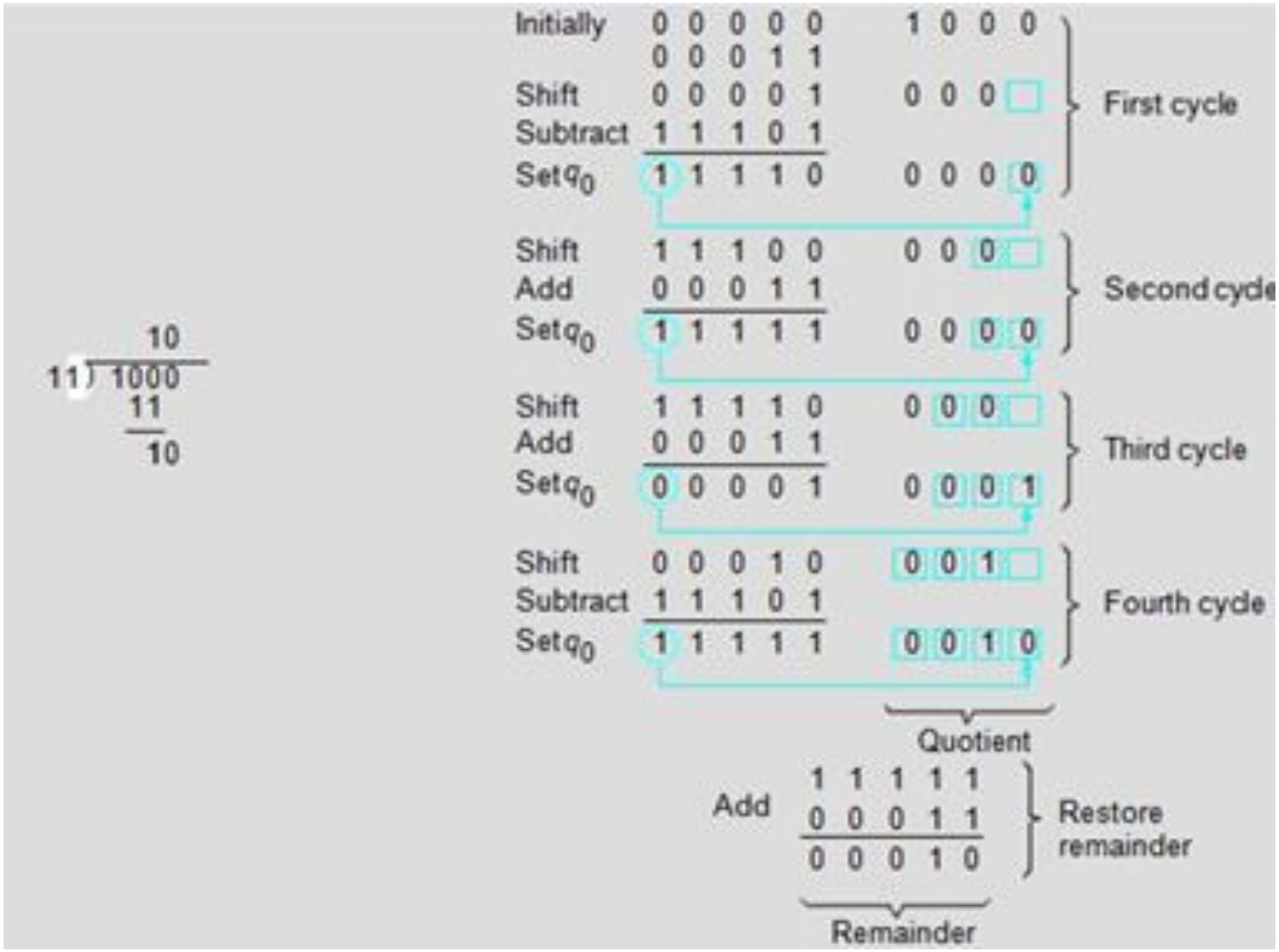
Non-Restoring Division

Restoring division can be improved using non-restoring algorithm

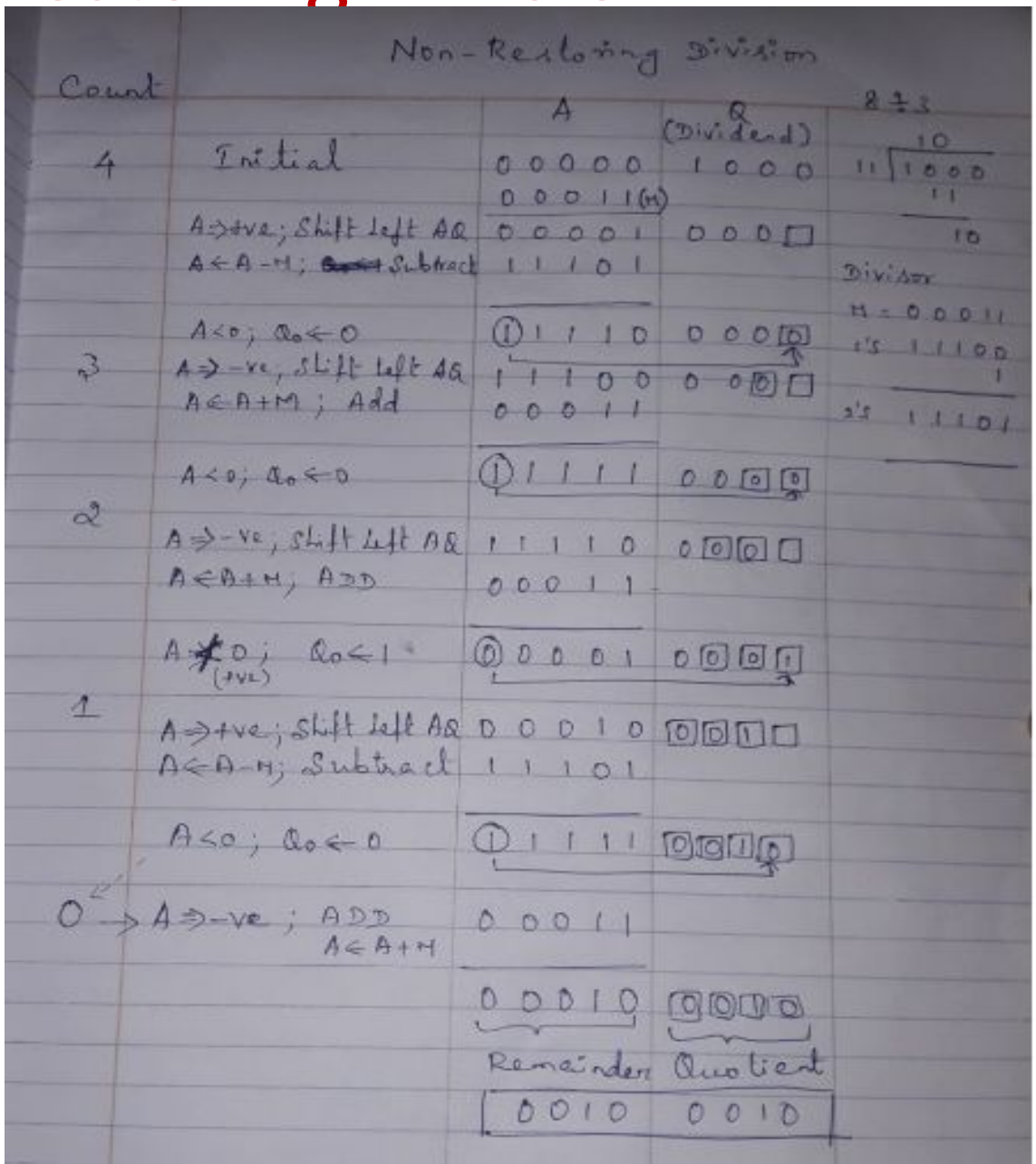
The effect of restoring algorithm actually is:
If A is positive, we shift it left and subtract M, that is compute $2A-M$
If A is negative, we restore it $(A+M)$, shift it left, and subtract M, that is, $2(A+M)-M = 2A+M$.
Set q_0 to 1 or 0 appropriately.

Non-restoring algorithm is:
Set A to 0.
Repeat n times:
  If the sign of A is positive:
    Shift A and Q left and subtract M. Set q0 to 1.
  Else if the sign of A is negative:
    Shift A and Q left and add M. Set q0 to 0.
If the sign of A is 1, add A to M.

Non-Restoring Division



Non-Restoring Division



Non-Restoring Division

Count		A	Q (Dividend)
4	Initial	0 0 0 0 0	1 0 0 0
		0 0 0 1 1 (M)	
	A ⇒ +ve; Shift left AQ	0 0 0 0 1	0 0 0 □
	A ← A − M; Subtract	1 1 1 0 1	
	A < 0; $Q_0 \leftarrow 0$	(1) 1 1 1 0	0 0 0 0
3	A ⇒ −ve, Shift left AQ	1 1 1 0 0	0 0 0 □
	A ← A + M; Add	0 0 0 1 1	
	A < 0; $Q_0 \leftarrow 0$	(1) 1 1 1 1	0 0 0 0
2	A ⇒ −ve, Shift left AQ	1 1 1 1 0	0 0 0 □
	A ← A + M; ADD	0 0 0 1 1	
	A ≠ 0; $Q_0 \leftarrow 1$ (+VE)	(0) 0 0 0 1	0 0 0 1
1	A ⇒ +ve; Shift left AQ	0 0 0 1 0	0 0 1 □
	A ← A − M; Subtract	1 1 1 0 1	
	A < 0; $Q_0 \leftarrow 0$	(1) 1 1 1 1	0 0 1 0
0 →	A ⇒ −ve; ADD A ← A + M	0 0 0 1 1	
		0 0 0 1 0	0 0 1 0
		Remainder	Quotient
		0 0 1 0	0 0 1 0

8 ÷ 3

```
    10
11 | 1000
     11
     --
     10
```

Divisor
M = 0 0 0 1 1
1's 1 1 1 0 0
+ 1
2's 1 1 1 0 1

Floating Point Numbers and Operations

Floating Point Number Representation (FPR)

$$\pm \text{Significant} \times \text{Base}^{\pm \text{Exponent}}$$

Example: $+10.55 \times 10^{55}$

Why FPR?

- Consider a very small number 0.00000000005
- As it consists of many 0's after the decimal point, it requires more number bits for representation
- 0.00000000005 is represented as **0.5×10^{-10},** it will require only fewer bits
- Consider a very large number 50000000000 which can be represented as 5×10^{10} (+5
- **Consider the example $0.123 \times 10^4 = 0.0123 \times 10^5 = 123 \times 10^3$**
- But we need a fixed and single representation for floating point numbers. For this normalization is required.

Floating Point Numbers and Operations

Normalization rules for floating point number

There are 2 rules:

(1) The integer part should be 0

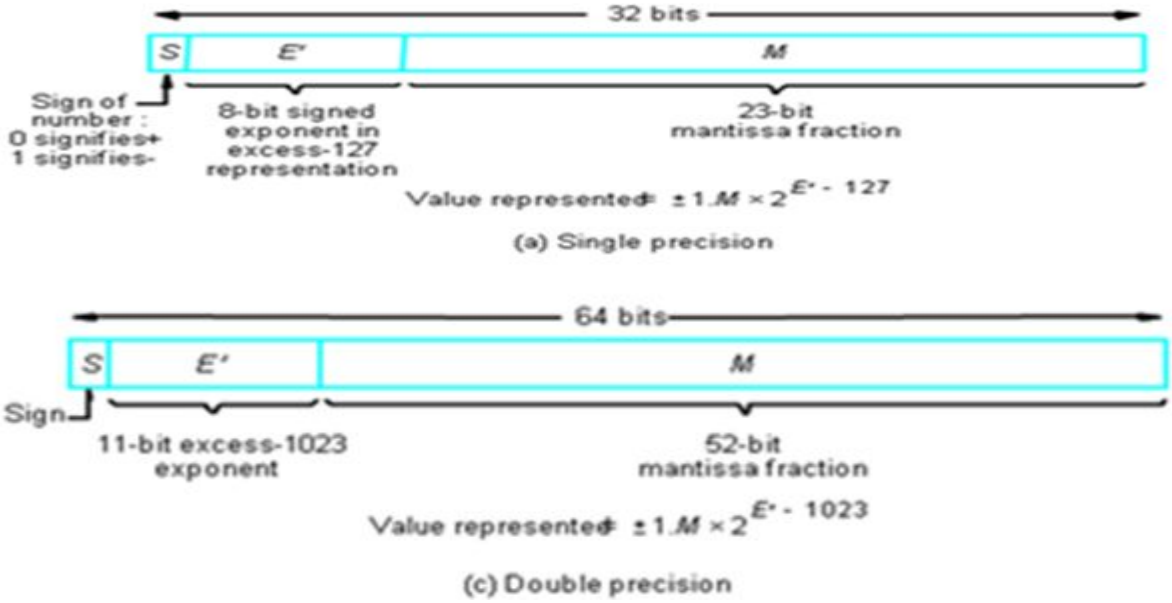
(2) $0.d_1d_2...d_n \times B^{\pm E}$ then $d_1 > 0$ and all $\sum_{i=2}^{n} d_i \geq 0$

In the example **0.123×10^4 = ~~0.0123×10^5~~ = ~~123×103,~~ the strikethrough representations are wrong with respect to normalization rules.**

Two floating point representation techniques are,

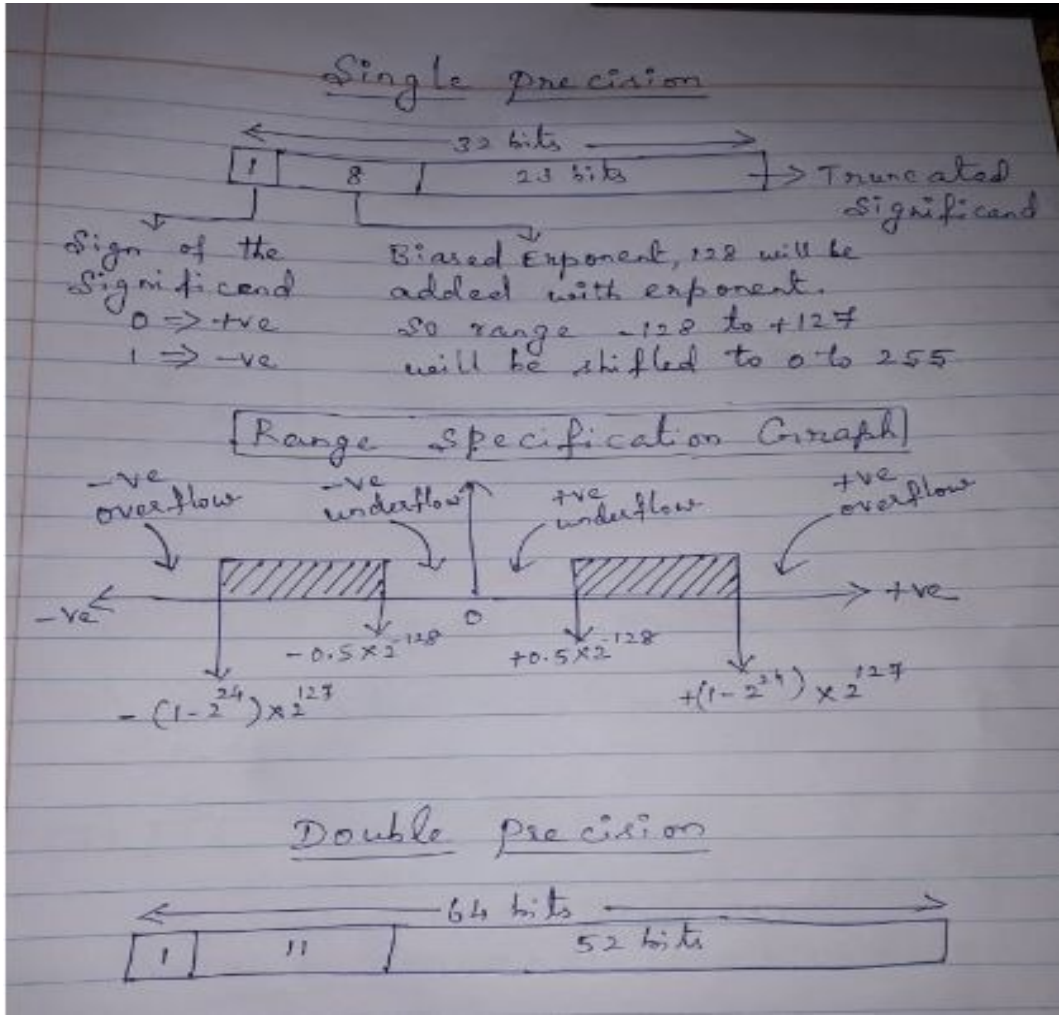
(1) Single precision (32 bits or 4 bytes)

(2) Double precision (64 bits or 8 bytes)



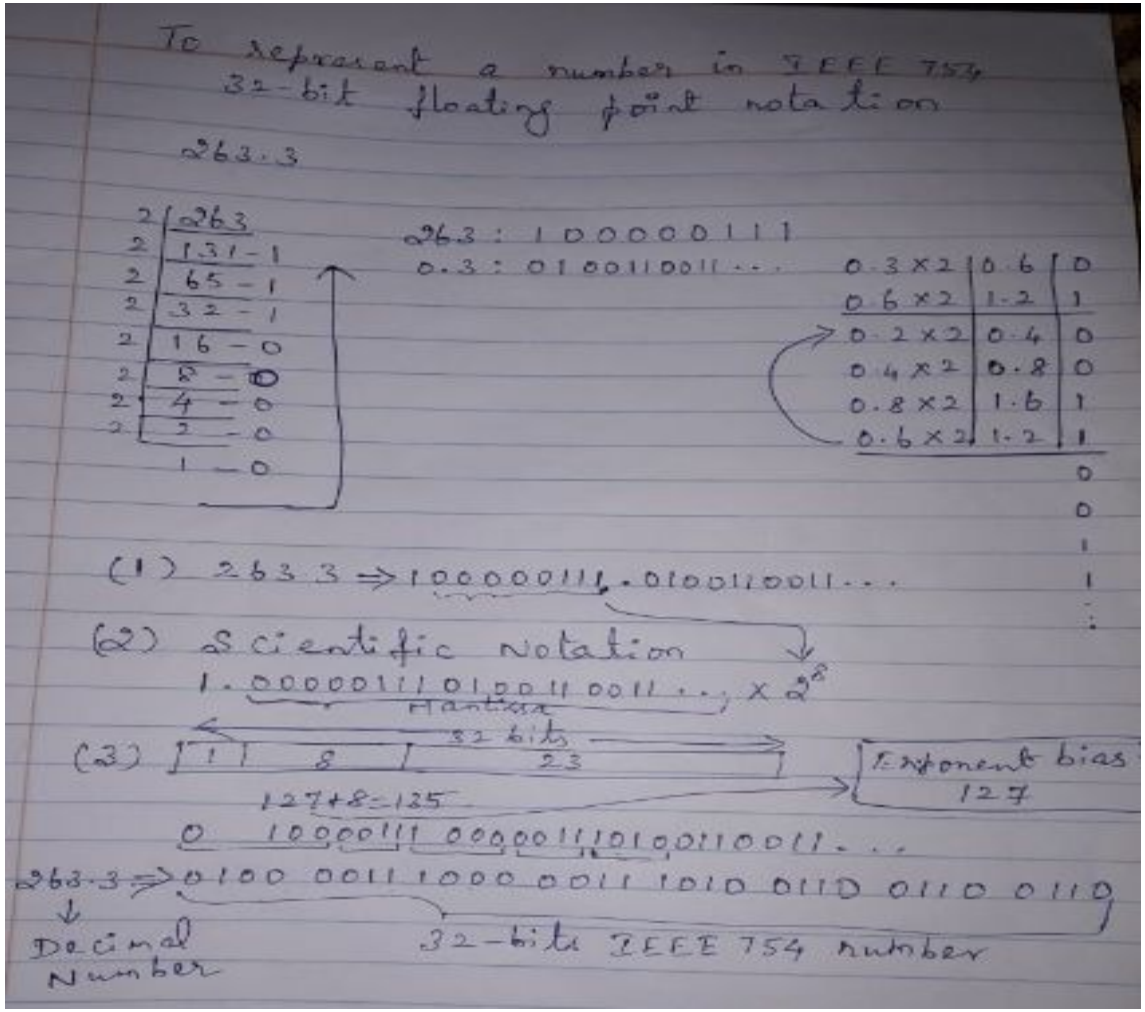
Floating Point Numbers and Operations

Single Precision and Double Precision Format



Floating Point Numbers and Operations

Decimal number into IEEE 754 32-bit floating point number



Floating Point Numbers and Operations

Floating Point Arithmetic Addition/Subtraction

Steps to add/subtract two floating point numbers:

1. Compare the magnitudes of the exponents and make suitable alignment to the number with the smaller magnitude of exponent
2. Perform the addition/subtraction
3. Perform normalization by shifting the resulting mantissa and adjusting the resulting exponent

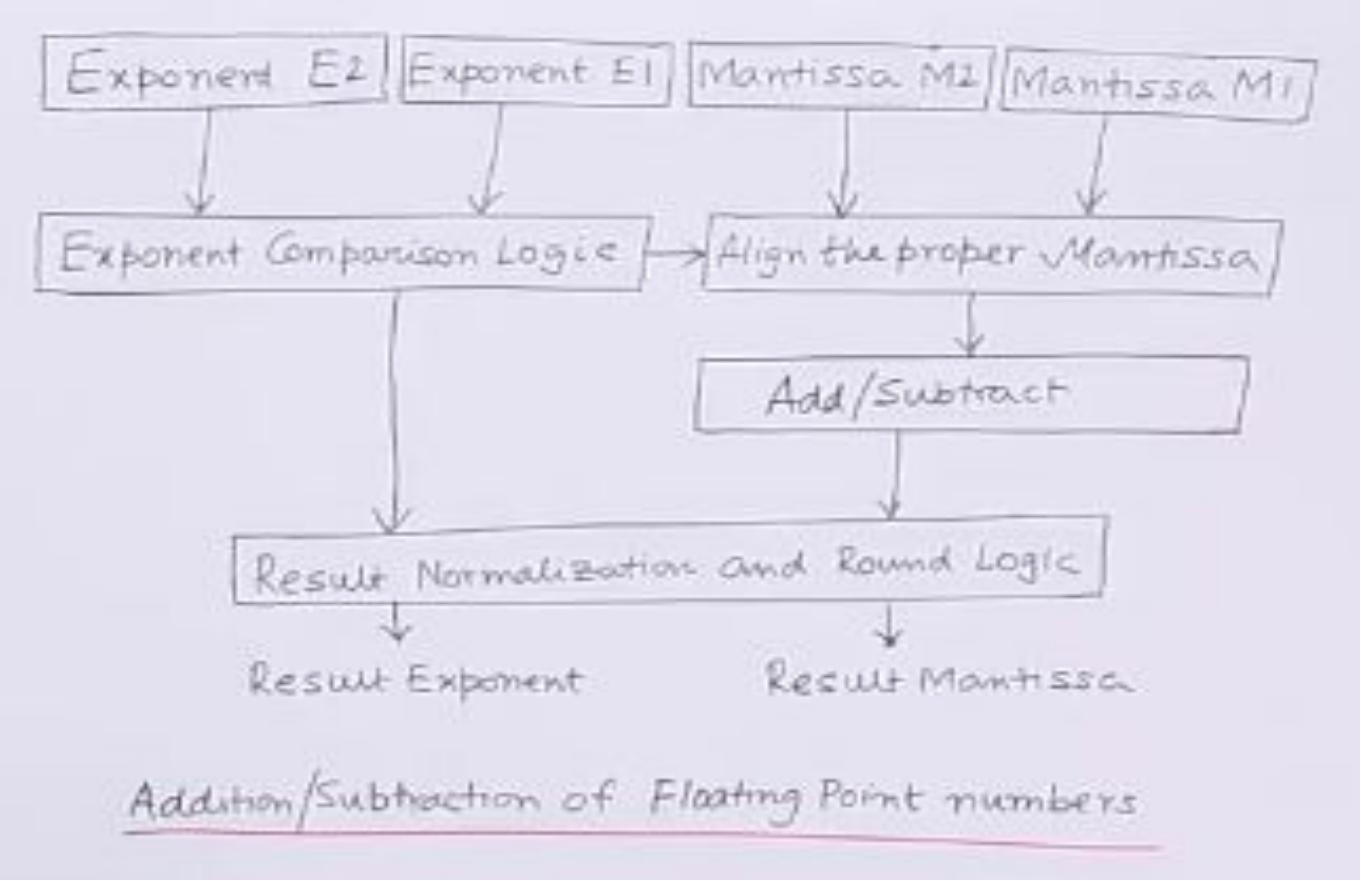
Example: Add 1.1100×2^4 and 1.1000×2^2

1. Alignment: 1.1000×2^2 has to be aligned to 0.0110×2^4 1.1100
2. Addition: Add the numbers to get 10.0010×2^4 0.0110
3. Normalization: Find normalized result 10.0110

0.1000×2^6 (assuming 4-bits are allowed after decimal point)

Floating Point Numbers and Operations

Floating Point Arithmetic Addition/Subtraction



Reference Links

Ripple Carry Adder

- https://www.gatevidyalay.com/tag/advantages-of-ripple-carry-adder/

Floating Point Numbers and Operations

- https://www.youtube.com/watch?v=XOMTNy2qiZ0&t=72s
- https://www.youtube.com/watch?v=8afbTaA-gOQ
- https://www.youtube.com/watch?v=w7NQTb1FTDU