Unit 3

ALU.

DAY: A B C

DATE: / /

1. Addition of positive numbers.
2. Fast adders.
3. signed addition and subtraction.
4. addition/subtraction Logic unit.
5. multiplication of positive numbers.
6. Array multiplier.
7. Sequential multiplier.
8. Signed number multiplication. 3 φ 1
9. multiplication using Booth's algorithm.
10. Fast multiplication.
11. Bit pair recording of multiplication.
12. Division restoring and non restoring algorithms.
13. Floating point numbers and operations.

Teacher's Signature.....

1. Addition of positive numbers:

$$\begin{array}{r} 0 \\ +\,0 \\ \hline 0 \end{array} \qquad \begin{array}{r} 1 \\ +\,0 \\ \hline 1 \end{array} \qquad \begin{array}{r} 0 \\ +\,1 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ +\,1 \\ \hline 10 \end{array}$$

↑ carry out

Adder:

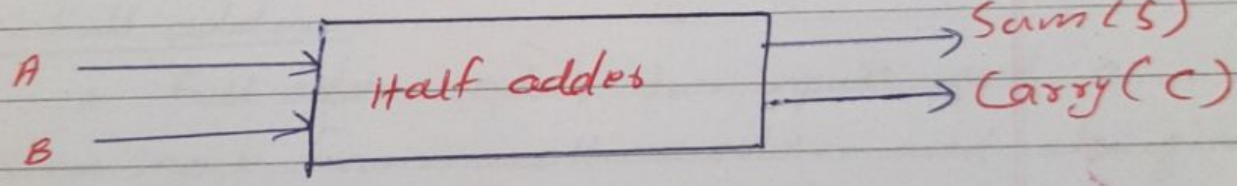
An adder is a device that will add together two bits and give the result as the output.

There are two kinds of adders:

1. Half adder
2. Full adder

1. Half adder:

It adds two bits together and gives a two bit output.



Truth table:

A	B	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$sum = \bar{A}B + A\bar{B}$

$= A \oplus B$

$carry = AB$

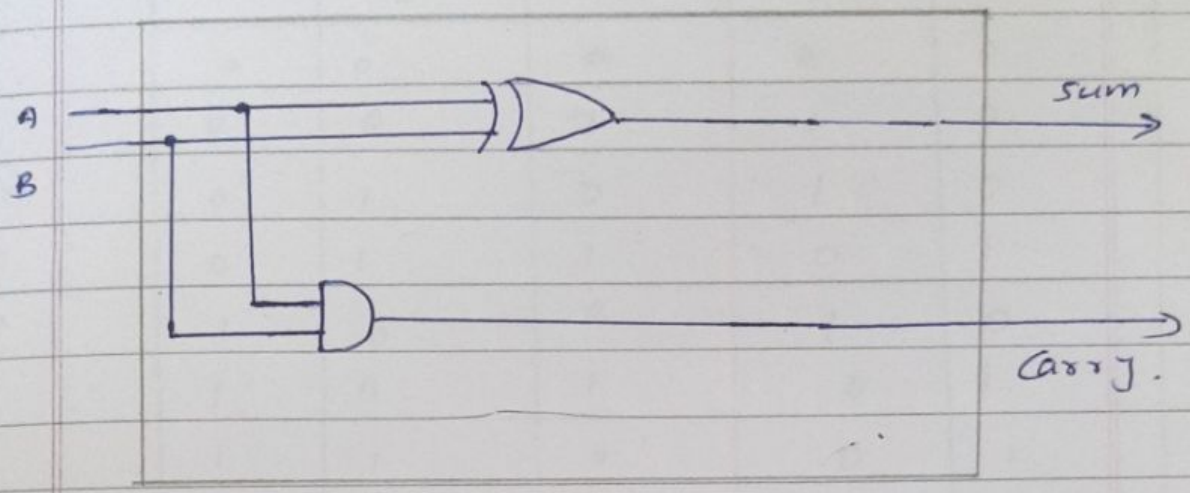
Teacher's Signature.....

A B C

DAY:

DATE: / /

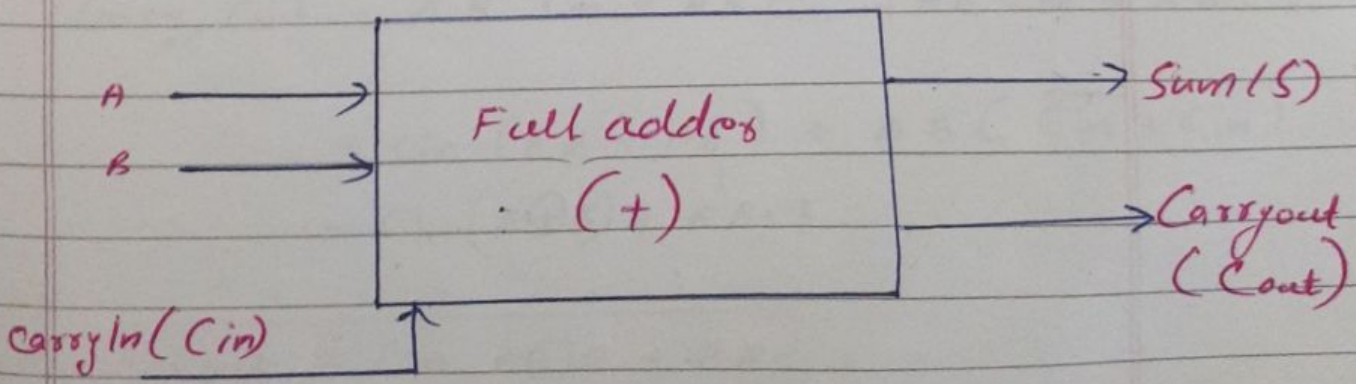
Logic diagram:



Half adders have no scope of adding the carry bit resulting from the addition of previous bits. To over come this drawback, full adder comes into play.

2. Full adder:

A full adder adds two inputs and a carried input from another adder and also gives a two-bit output.



Teacher's Signature.....

A B C

DAY :

DATE : / /

Truth table:

A	B	C_{in}	Sum	Cout
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

$$Sum = \bar{A}\,\bar{B}\,C_{in} + \bar{A}\,B\,\bar{C}_{in} + A\,\bar{B}\,\bar{C}_{in} + A\,B\,C_{in}$$

$$= \bar{A}(\bar{B}C_{in} + B\,\bar{C}_{in}) + A(\bar{B}\,\bar{C}_{in} + B\,C_{in})$$

$$= \bar{A}\,(B \oplus C_{in}) + A(\overline{B \oplus C_{in}})$$

$$\boxed{Sum = A \oplus B \oplus C_{in}} \quad [\because a \oplus b = \bar{a}b + a\bar{b}]$$

$$Carry,\ Cout = \bar{A}\,B\,C_{in} + A\bar{B}\,C_{in} + AB\,\bar{C}_{in} + AB\,C_{in}$$

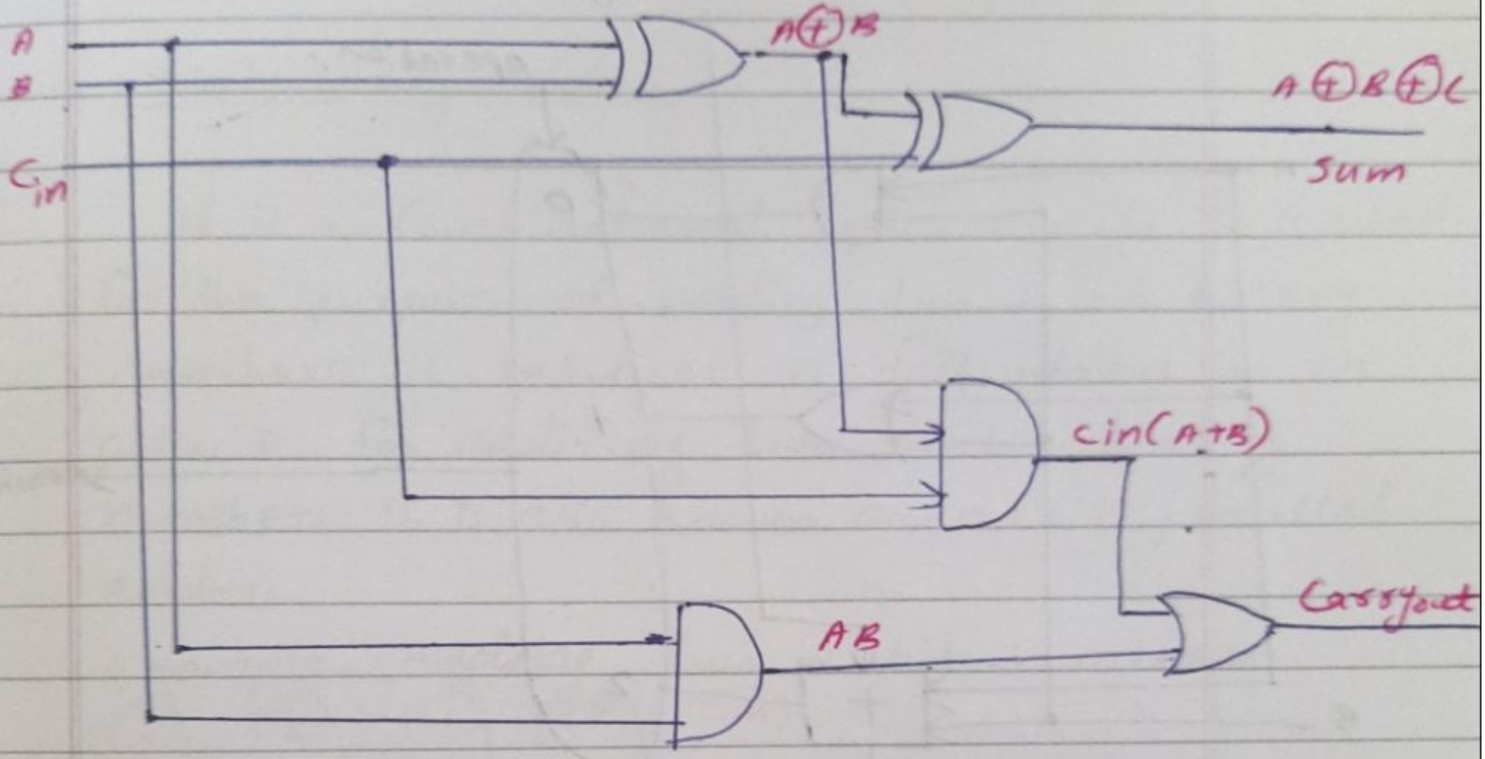
$$= C_{in}(\bar{A}B + A\bar{B}) + AB(\overline{C_{in}} + C_{in})$$

$$= C_{in}(A \oplus B) + AB \cdot 1$$

$$\boxed{Cout = C_{in}\,A \oplus B + AB}$$

Teacher's Signature.....

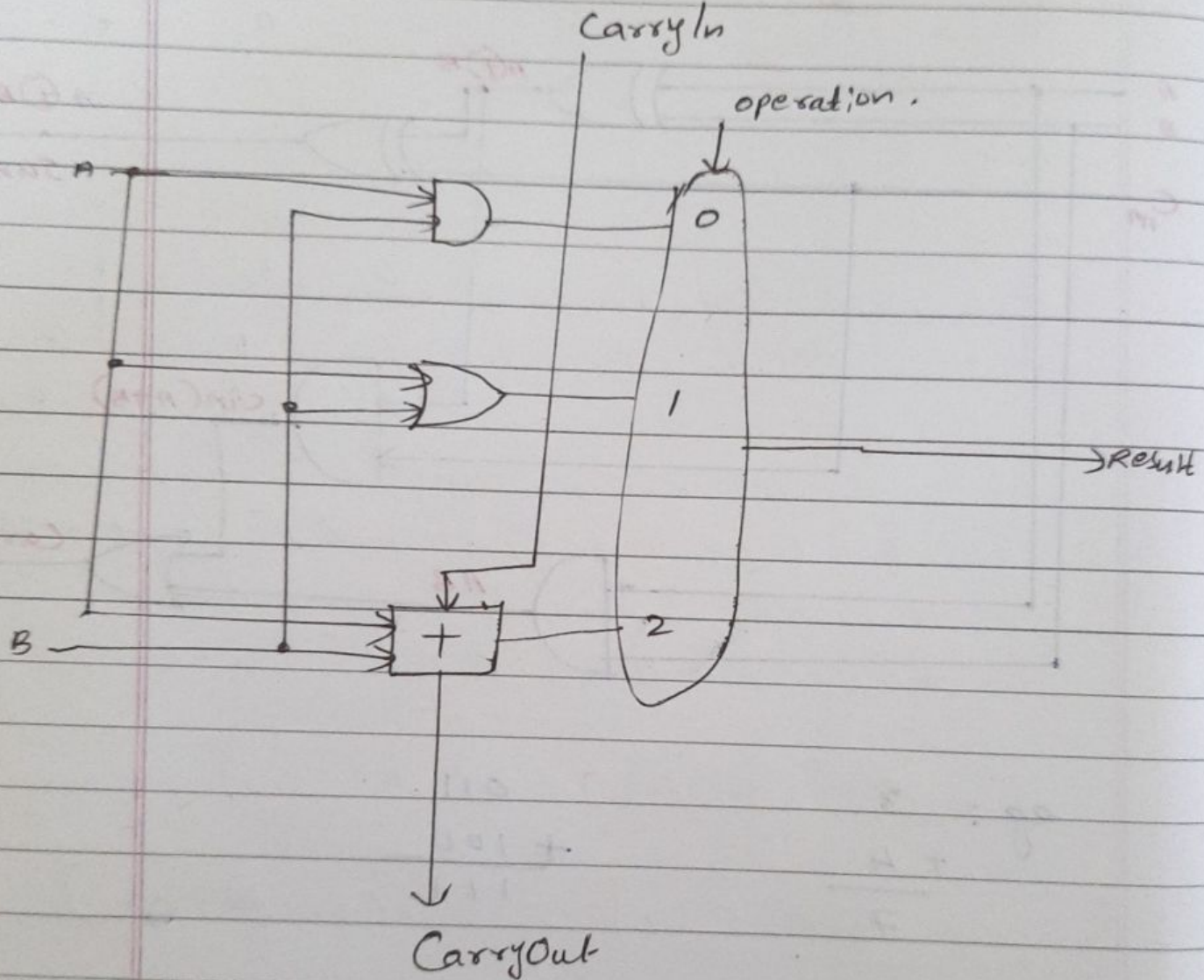
Logic diagram:



eg: 3 + 4 = 7

011 + 100 = 111

1 bit ALU that performs AND, OR and addition:



A B C

DAY:

DATE: / /

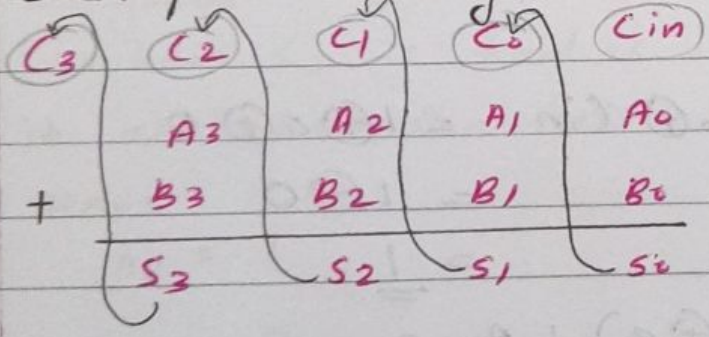
2. Fast adders:

The ripple carry adder and look ahead carry adder will speed up the addition process.

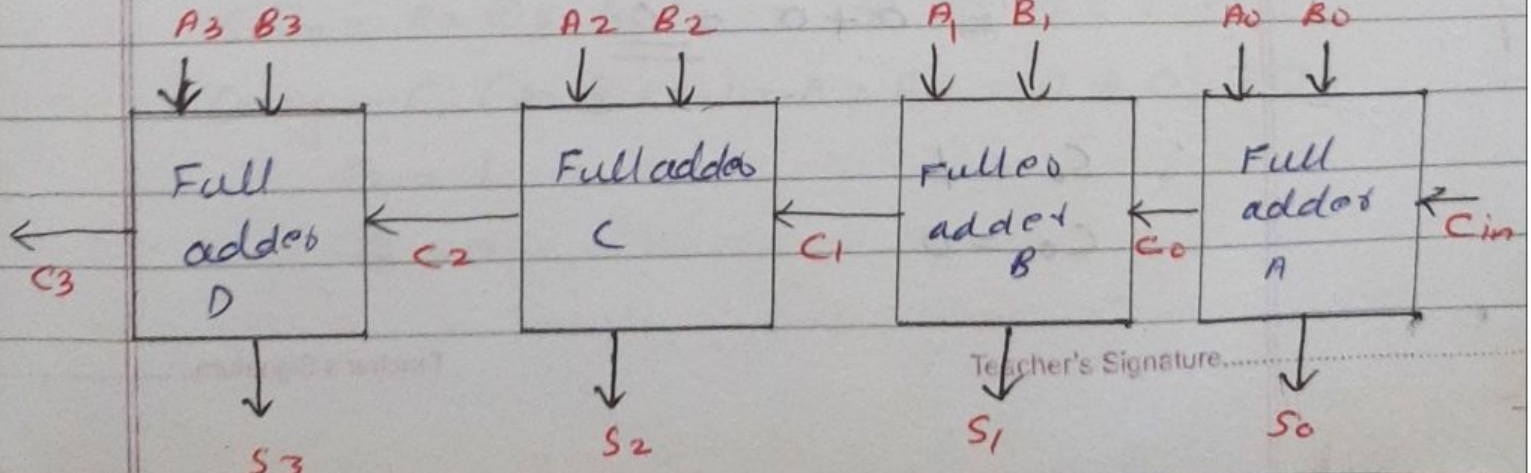
• Ripply carry adder:

It is a Combinational logic circuit. It is used for the purpose of adding two n bit binary numbers. It requires n full adders in its circuit for additing two n-bit binary numbers. It is also known as n-bit parallel adder.

Example: Adding two 4 bit numbers.



using ripple Carry adder, this addition is carried out as shown by the following logic diagram:



Teacher's Signature.....

A B C

DAY:

DATE: / /

Working of 4 bit ripple carry Adder:

Let

Two 4 bit numbers are:

0 1 0 1 (A_3 A_2 A_1 A_0)

1 0 1 0 (B_3 B_2 B_1 B_0)

These numbers are to be added using a 4 bit ripple carry adder.

Stage 1:

When Cin is fed as input to the full adder A, it activates the full adder A.

Here

$A_0 = 1$

$B_0 = 0$

$C_{in} = 0$

Compute:

$Sum = A_0 \oplus B_0 \oplus C_{in} = 1 \oplus 0 \oplus 0$

$= 1 \oplus 0$

$= \underline{1}$

$Carry = C_{in}(A_0 \oplus B_0) + A_0 B_0$

$= 0 (1 \oplus 0) + 1.0$

$= 0 + 0 = \underline{0}$

$\therefore S_0 = 1$

$C_0 = 0$

B	A⊕B
0	0
1	1
0	1
1	0

Teacher's Signature.....

Stage 2:

When C_0 is fed as input to the full adder B, it activates the full adder B.

Here

$A_1 = 0$

$B_1 = 1$

$C_0 = 0$

Computes:

$\text{Sum} = A_1 \oplus B_1 \oplus C_0 = 0 \oplus 1 \oplus 0 = 1$

$\text{Carry} = C_0 (A_1 \oplus B_1) + A_1 \cdot B_1 = 0 + 0 = 0$

$\therefore S_1 = 1$

$C_1 = 0$

Stage 3:

When C_1 is fed as input to the full adder C, it activates the full adder C.

Here

$A_2 = 1$

$B_2 = 0$

$C_1 = 0$

Computes

$\text{Sum} = A_2 \oplus B_2 \oplus C_1 = 1 \oplus 0 \oplus 0 = 1$

$\text{Carry} = C_1 (A_2 \oplus B_2) + A_2 B_2 = 0 + 0 = 0$

$\therefore S_2 = 1$

$C_2 = 0$

Teacher's Signature.....

Stage 4:

When C_2 is fed as input to the full adder D, it activates the full adder D.

Here

$A_3 = 0$

$B_3 = 1$

$C_2 = 0$

Computes:

$Sum = A_3 \oplus B_3 \oplus C_2 = 0 \oplus 1 \oplus 0 = 1$

$Carry = C_2(A_3 \oplus B_3) + A_3B_3 = 0 + 0 = 0$

$\therefore$

$S_3 = 1$

$C_3 = 0$

Thus finally outputs:

$Sum = S_3 S_2 S_1 S_0 = 1111$

$Carryout = C_3 = 0$

In ripple carry adder, each full adder has to wait for its carry-in from its previous stage full adder. Thus, n^{th} full adder has to wait until all $(n-1)$ full adders have completed their operations. This causes a delay and

makes ripple carry adder extremely slow. The situation becomes worst when the value of n becomes very large. To overcome this disadvantage, carry look ahead adder comes into play.

• Carry Look ahead adder:

It is an improved version of the ripple carry adder. It generates the carry-in of each full adder simultaneously without causing any delay.

This is the addition technique that eliminates the problem due to interstage carry delay. The lookahead carry addition will therefore speed up the addition process.

Design of carry look ahead adder:

To reduce the computation time, there are faster ways to add two binary number by using carry look ahead adders. they work by creating two signals P and G known to be carry propagator and carry generater.

The carry Propagator is propagated to the next level where as the carry generator is used to generate the output carry, regardless of input carry.

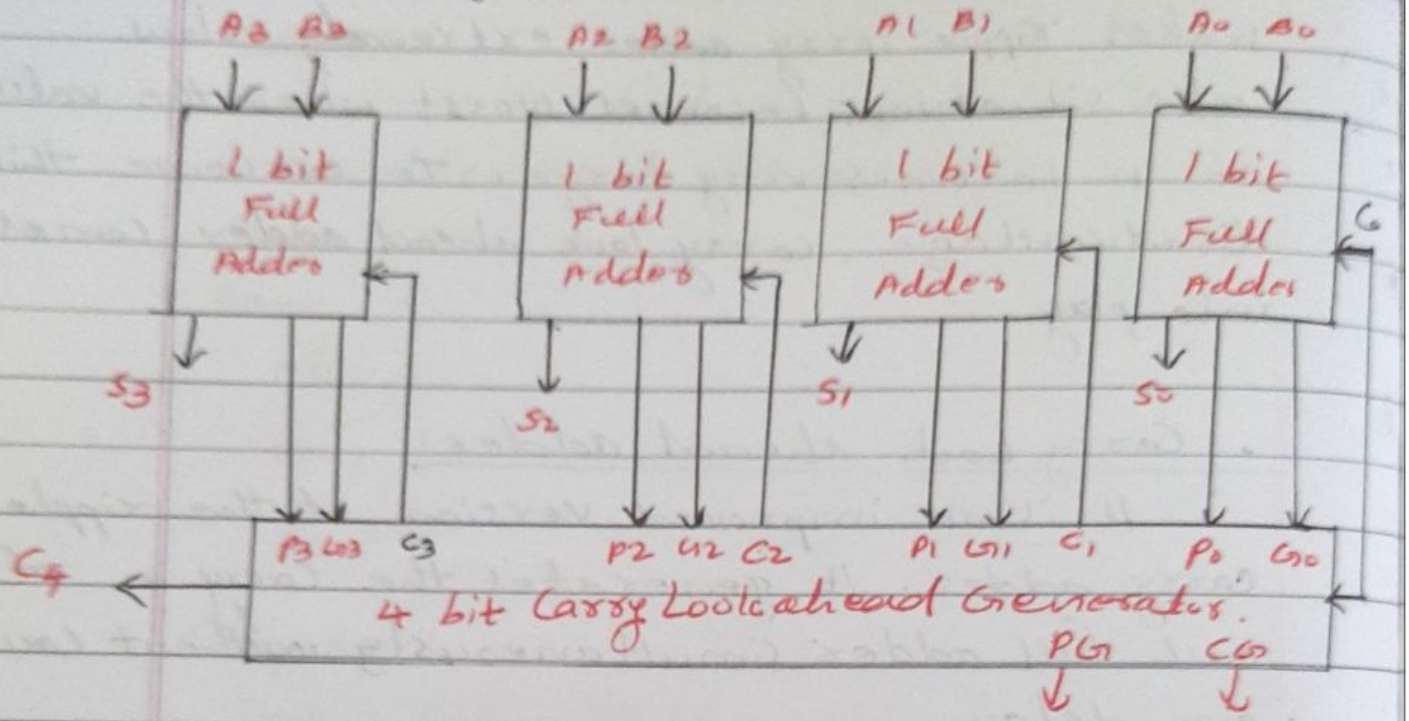
Teacher's Signature.....

A B C

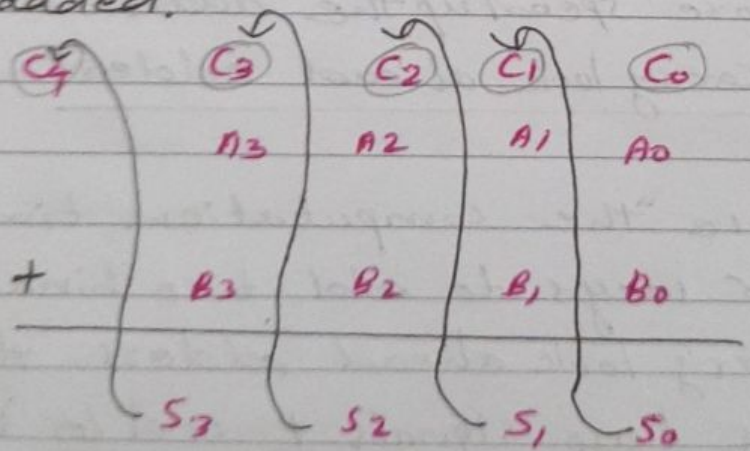
DAY:

DATE: / /

Block Diagram:



Consider two 4 bit binary numbers $A_3 A_2 A_1 A_0$ and $B_3 B_2 B_1 B_0$ are to be added.



From here, we have

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

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

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

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

Teacher's Signature.....

Let

$G_i = A_i B_i$ where G is called carry generator.

$P_i = A_i \oplus B_i$ where P is called carry propagator.

Then, rewriting the above equations, we have,

$$C_1 = C_0 P_0 + G_0 \quad \text{---} (1)$$

$$C_2 = C_1 P_1 + G_1 \quad \text{---} (2)$$

$$C_3 = C_2 P_2 + G_2 \quad \text{---} (3)$$

$$C_4 = C_3 P_3 + G_3 \quad \text{---} (4)$$

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

Finally, we have the following equations.

$$C_1 = C_0 P_0 + G_0$$

$$C_2 = C_0 P_0 P_1 + G_0 P_1 + G_1$$

$$C_3 = C_0 P_0 P_1 P_2 + G_0 P_1 P_2 + G_2 P_2 + G_2$$

$$C_4 = C_0 P_0 P_1 P_2 P_3 + G_0 P_1 P_2 P_3 + G_2 P_2 P_3 + G_3 P_3 + G_3$$

These equations show that the carry-in of any stage full adder.

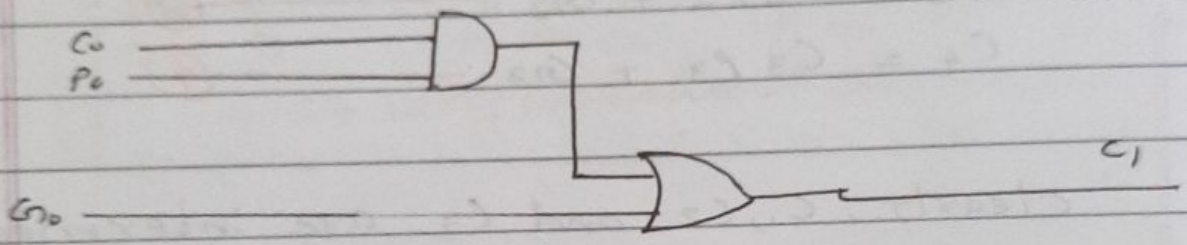
Teacher's Signature.....

DAY :
DATE : / /

Implementation of Carry Generator Circuit:

- Implementation of C_1:

$$C_1 = C_0 P_0 + G_0$$



similarly implement C_2, C_3 and C_4.

Advantage:

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

Teacher's Signature.....

A B C

DAY :

DATE : / /

3. Signed addition and subtraction:

We designate the magnitude of the two numbers by A and B. Where the signed numbers are added or subtracted, we find that there are 8 different conditions to consider, depending on the sign of the numbers and the operation performed.

operation	Add magnitude	subtract magnitude		
		when A>B	when A<B	when A=B
$(+A)+(+B)$	$+(A+B)$	—	—	—
$(+A)+(-B)$	—	$+(A-B)$	$-(B-A)$	$+(A-B)$
$(-A)+(+B)$	—	$-(A-B)$	$+(B-A)$	$+(A-B)$
$(-A)+(-B)$	$-(A+B)$	—	—	—
$(+A)-(+B)$	—	$+(A-B)$	$-(B-A)$	$+(A-B)$
$(+A)-(-B)$	$+(A+B)$			
$(-A)-(+B)$	$-(A+B)$			
$(-A)-(-B)$	—	$-(A-B)$	$+(B-A)$	$+(A-B)$

Addition & subtraction of signed numbers.

Teacher's Signature.....

A B C

DAY:

DATE: / /

Addition:

A + B

A : Augend

B : Addend

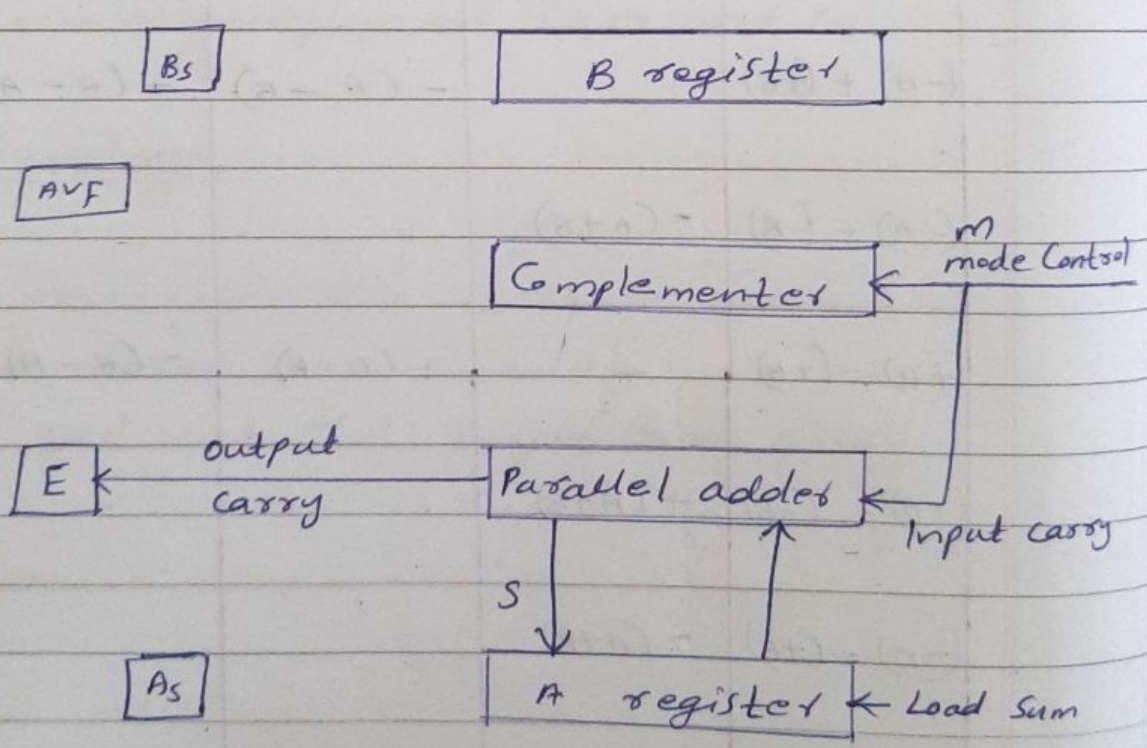
Subtraction

A − B

A : minuend

B : subtrahend

Hardware Implementation:



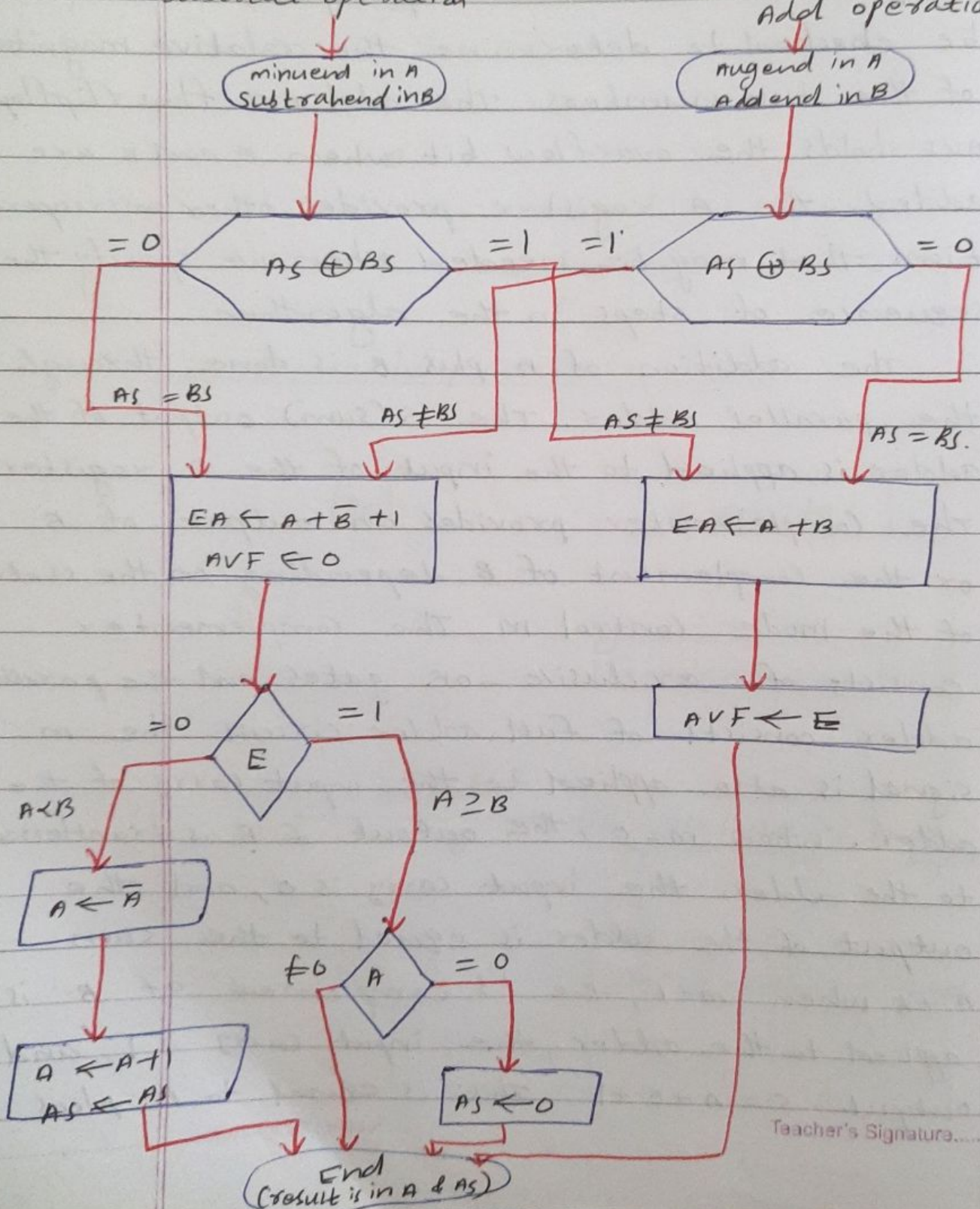
Teacher's Signature.....

It consist of registers A and B and sign flip flops A_s and B_s. subtraction is done by adding A to the 2's complement of B. The output carry is transferred to flip flop E, where it can be checked to determine the relative magnitudes of the two numbers. The add overflow flipflop AVF holds the overflow bit when A and B are added. the A register provides other microoperations that may be needed when we specify the sequence of steps in the algorithm.

The addition of A plus B is done through the parallel adder. The S (sum) output of the adder is applied to the input of the A register. The complementer provides an output of B or the complement of B depending on the state of the mode control M. The complementer consists of exclusive-OR gates and the parallel adder consists of full adder circuit. the m signal is also applied to the input carry of the adder. when $m=0$, the output of B is transferred to the adder, the input carry is 0, and the output of the adder is equal to the sum $A+B$ when $m=1$, the 1's complement of B is applied to the adder, the input carry is 1, and output $S = A+B+1$. This is equal to A plus

the 2's Complement of B, whis is equivalent to the subtraction A − B.

Flowchart:



A B C

DAY:

DATE: / /

4. addition/subtraction Logic unit:

Refer module 2

performing arithmetic or logic operations.

Teacher's Signature.....

5. multiplication of positive numbers:

multiplication of two k bit number needed multioperand addition process that can be realized in k cycles of shifting and addition with hardware.

mxN bit multiplier requires m+N bits result : $2^m \times 2^N = 2^{M+N}$

Consider a 4 bit x 4 bit binary multiplier:

multiplicand 1 1 0 0_2 = 12

multiplier 1 1 0 0_2 = 13

---

1 1 0 0

0 0 0 0

1 1 0 0

1 1 0 0

---

product 1 0 0 1 1 1 0 0_2 = 156

with only two choices, each step of the multiplication is simple.

1) 0 x multiplicand = 0

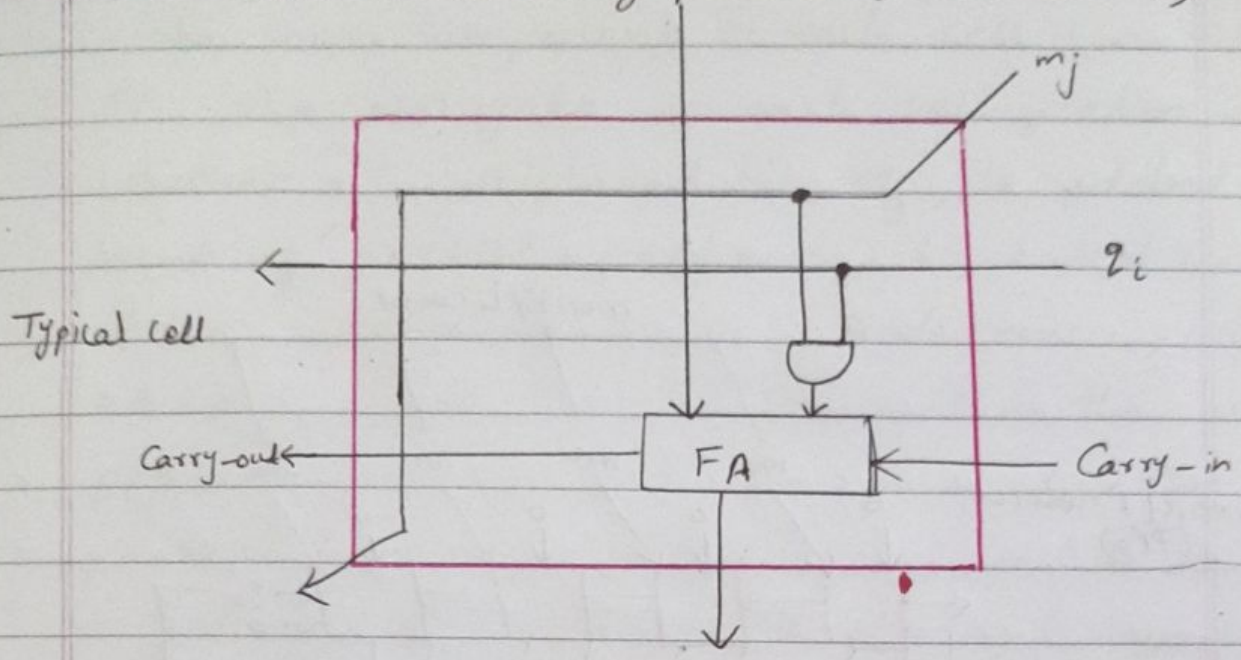
2) 1 x multiplicand = multiplicand.

5.1. Array multipliers:

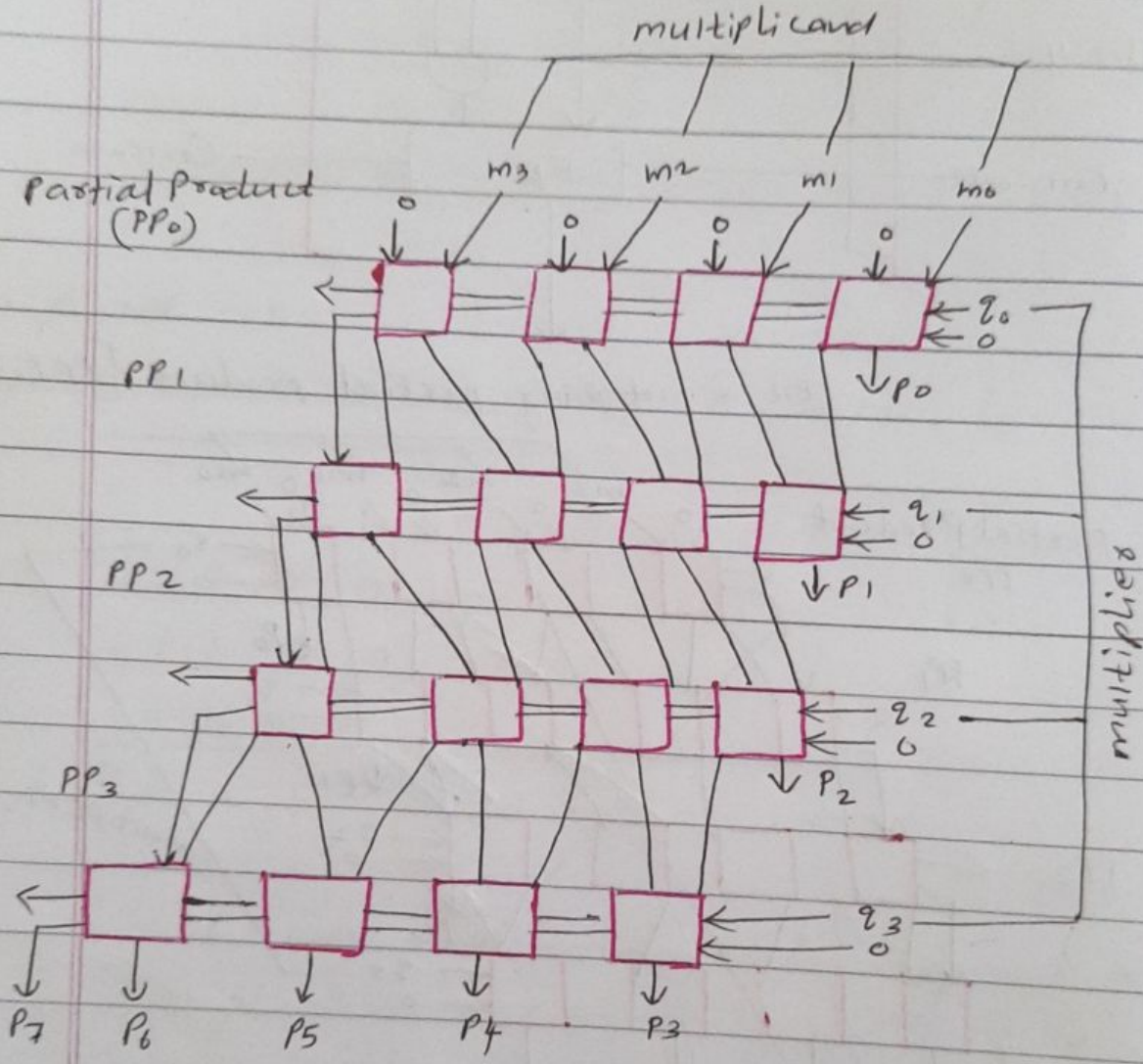
Binary multiplication of positive operands can be implemented in a combinational, two dimensional logic array.

Teacher's Signature.....

bit of incoming partial product (PP_i)



Bit of ~~out~~going partial product [PP(i+1)]



$PP_4 = P_7, P_6, \ldots P_0 = \text{Product}$.

Array Implementation.

Teacher's Signature.....

The main component in each cell is a full adder FA. The AND gate in each cell determines whether a multiplicand bit, m_j, is added to the incoming partial-product bit, based on the value of the multiplier bit q_i. Each row i, where $0 \leq i \leq 3$, adds the multiplicand to the incoming partial product, PP_i, to generate the outgoing partial product, $PP(i+1)$, if $q_i = 1$. If $q_i = 0$, PP_i is passed vertically downward unchanged. PP_0 is all 0s, and PP_4 is the desired product. The multiplicand is shifted left one position per row by the diagonal signal path.

The worst case signal propagation delay path is from the upper right corner of the array to the high-order product bit output at the bottom left corner of the array. The path consists of the staircase pattern that includes the two cells at the right end of each row, followed by all the cells in the bottom row.

Assuming that there are two gate delays from the inputs to the outputs of a full adder block, the path has a total of $6(n-1)+1$ gate delays, including the initial AND gate delay in all cells, for the $n \times n$ array. Only the AND gates

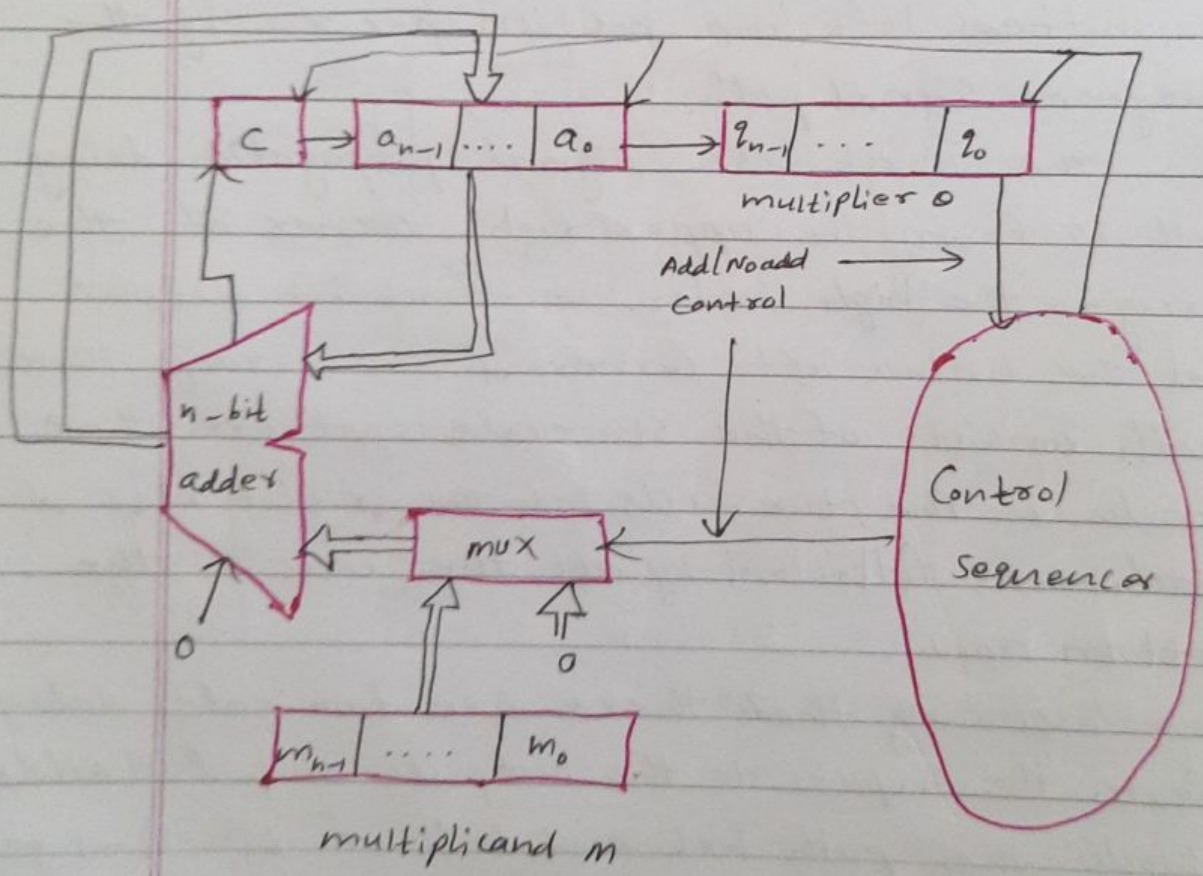
Teacher's Signature.....

are actually needed in the first row of the array because the incoming partial product PP_0 is zero.

5.2 Sequential multiplier:

The simplest way to perform multiplication is to use the adder circuitry in the ALU for a number of sequential steps.

Block Diagram:



The sequential multiplier circuit performs multiplication by using a single n-bit adder n times to implement the spatial addition performed by the n rows of ripple-carry adders.

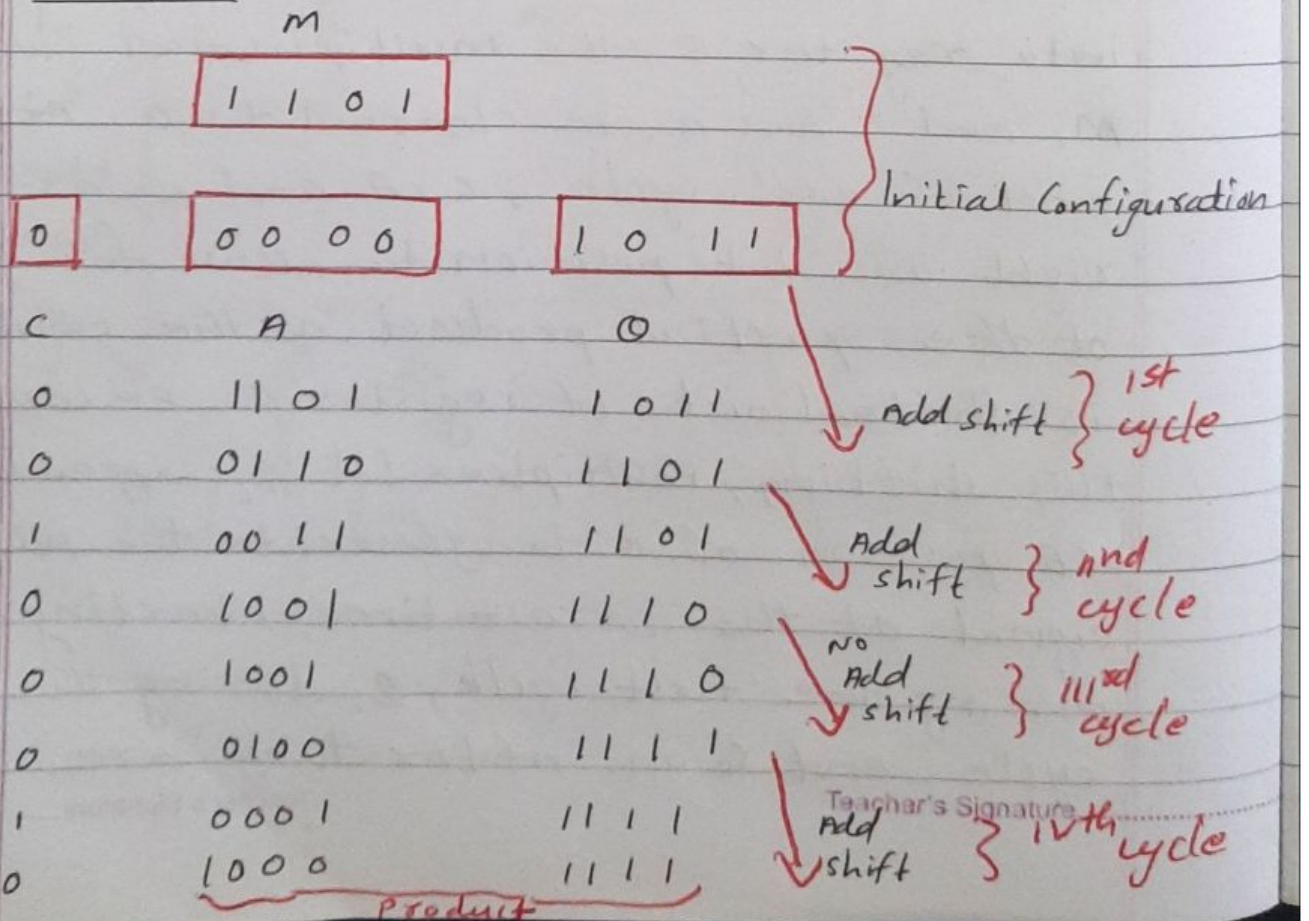
Registers A and Q combined hold PP_i while multiplier bit q_i generates the signal Add/Noadd. This signal controls the addition of the multiplicand, M, to PP_i to generate $PP(i+1)$. The product is computed in n cycles. The partial product grows in length by one bit per cycle from the initial vector, PP_0, of n 0s in register A. The carry-out from the adder is stored in flip flop C, shown at the left end of register A. At the start, the multiplier is loaded into register Q, the multiplicand into register M, and C and A are cleared to 0. At the end of each cycle, C, A and Q are shifted right one bit position to allow for growth of the partial product as the multiplier is shifted out of register Q. Because of this shifting, multiplier bit q_i appears at the LSB position of Q to generate the Add/No add signal at the correct time, starting with q_0 during the first cycle, q_1 during the second cycle, and so on. After they are used,

Teacher's Signature.....

the multiplier bits are discarded by the right-shift operation. The carry-out from the adder is the leftmost bit of PP(i+1), and it must be held in the C flip-flop to be shifted right with the contents of A and Q. After n cycles, the high-order half of the product is held in register A and the low order half is in register Q.

Using this sequential hardware structure, it is clear that a multiply instruction takes much more time to execute than an Add instruction.

Example:



signed number multiplication

The multiplication of 2's complement signed operands, generating a double length product. The general strategy is still to accumulate partial products by adding versions of the multiplicand as selected by the multipliers bits.

Consider the case of a positive multiplier and a negative multiplicand. When we add a negative multiplicand to a partial product, we must extend the sign-bit value of the multiplicand to the left as far as the product will extend.

Eg:

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

For a negative multiplier, a straightforward solution is to form the 2's complement of both the multiplier and the multiplicand and proceed as in the case of a positive multiplier. This is possible because complementation of both

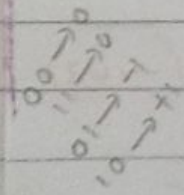
operands does not change the value or the sign of the product. A technique that works equally well for both negative and positive multipliers, called the Booth algorithm.

1. multiplication using Booth's algorithm :

The booth algorithm generates a 2n-bit product and treats both positive and negative 2's complement n-bit operands uniformly.

In the Booth scheme, -1 times the shifted multiplicand is selected when moving from 0 to 1, and +1 times the shifted multiplicand is selected when moving from 1 to 0, as the multiplier is scanned from right to left.

Eg: booth recording of a multiplier.



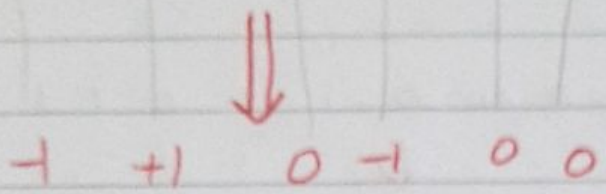
0 0 1 0 1 1 0 0 1 1 1 0 1 0 1 1 0 0

⇓

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

Teacher's Signature.....

Eg 2: 1 0 1 1 0 0

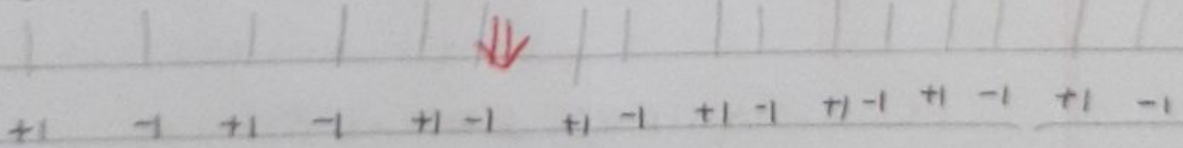


Booth multiplier recording table:

multiplier		Version of multiplicand selected by bit i
Bit i	Bit i-1	
0	0	0 × m
0	1	+1 × m
1	0	-1 × m
1	1	0 × m

Worst case multiplier:

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



In the worst case, that of alternating 1s and 0s in the multiplier,

ABC

DAY:

DATE: / /

ordinary multiplier:

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

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

good multiplier:

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

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

Booth multiplication with a negative multiplier:

Example:

multiplicand = +13 (0 1 1 0 1)

multiplier = -6 (1 1 0 1 0)

$$\begin{array}{r} 0\ 1\ 1\ 0\ 1 \\ \times\ 1\ 1\ 0\ 1\ 0 \\ \hline \end{array}$$

Teacher's Signature.....

1. 2's Complement of multiplicand.

0 1 1 0 1

↓ 1's complement +1

1 0 0 1 0 +

1

1 0 0 1 1

2. booth recording of a multiplier.

1 1 0 1 0

↓

0 -1 +1 -1 0

So recorded multiplier = 0 -1 +1 -1 0

3. multiplicand x recorded multiplier.

Note:

1. Recorded multiplier = -1
   partial product = 2's compliment of multiplicand
2. Recorded multiplier = 1
   partial product = multiplicand.
3. Recorded multiplier = 0
   partial product = 0.

```
                    0   1   1   0   1
                x   0  -1  +1  -1   0
              ---------------------------
    0   0   0   0   0   0   0   0   0   0
    1   1   1   1   1   0   0   1   1
    0   0   0   0   1   1   0   1
    1   1   1   0   0   1   1
    0   0   0   0   0   0
              ---------------------------
discard  ←  1   1   1   0   1   1   0   0   1   0   (-78)
carry
```

Eg2: multiply 3 x −7 using booth's algorithm.

solution:

multiplicand = 0011 (3)

multiplier = 2's complement of 7

0 111 (7)

2's complement of 7

```
  1000 +
     1
  ------
  1001
```

Teacher's Signature.....

A B C

DAY:

DATE: / /

1. 2's complement of multiplicand.

0 0 1 1

↓

1 1 0 0 +

1

1 1 0 1

2. Recorded multipliers:

1 0 0 1

↓

-1 0 +1 -1

3. multiplicand x recorded multipliers.

				0	0	1	1
			x	-1	0	+1	-1
1	1	1	1	1	1	0	1
0	0	0	0	0	1		
0	0	0	0	0	0		
+ 1	1	1	0	1			
1	1	0	1	0	0	1	1

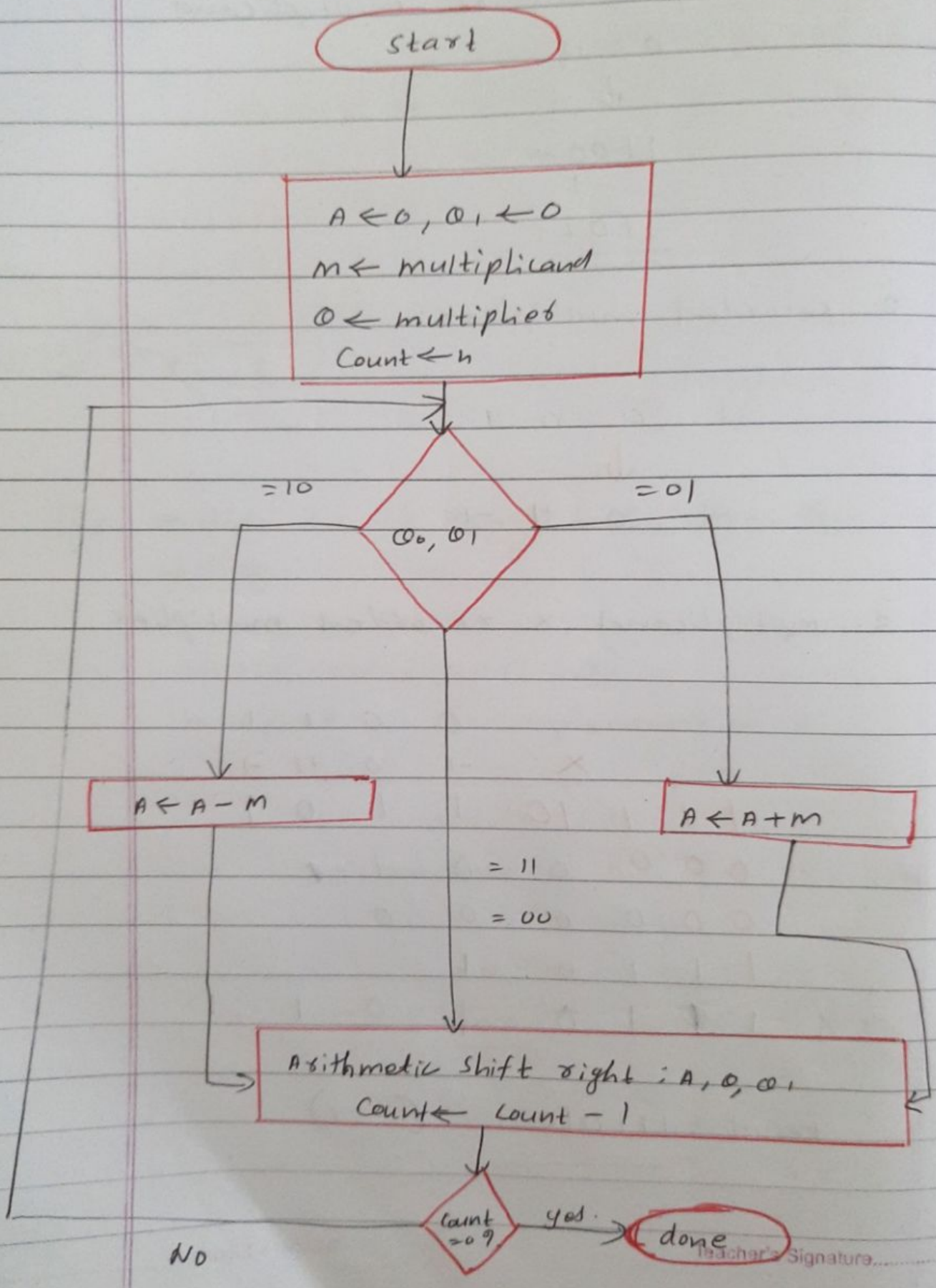
discard carry

Result = 1 1 1 0 1 0 1 1 (-21)

Teacher's Signature.....

DAY:
DATE: / /

algorithm:



Teacher's Signature.....

The booth algorithm has two attractive features:

1) It handles both positive and negative multipliers uniformly.
2) It achieves some efficiency in the number of additions required when the multiplier has a few large blocks of 1s.

Fast multiplication:

The bit-pair recording of multiplier technique is used for speeding up the multiplication operation. This technique guarantees that the maximum number of summands (versions of the multiplicand) that must be added is n/2 for n-bit operands.

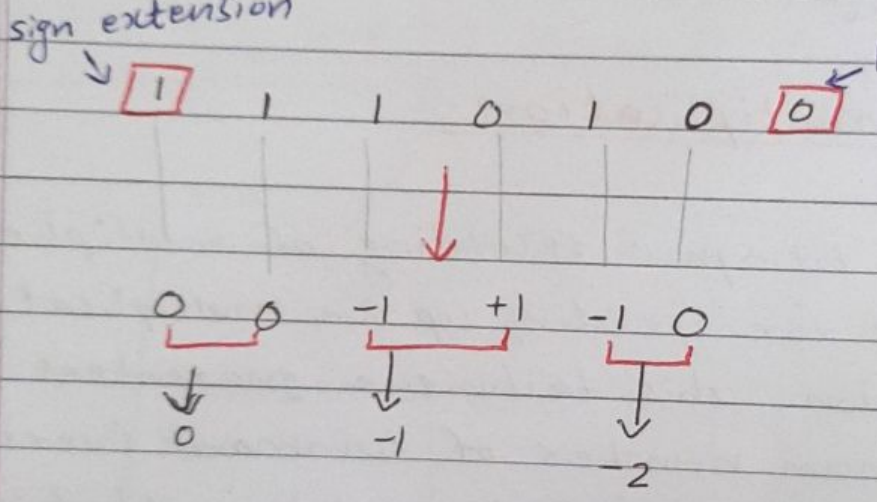
1. Bit-pair recoding of multiplication:

A technique called bit-pair recoding halves the maximum number of summands. It is derived directly 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). If the Booth-recoded multiplier is examined two bits at a

Teacher's Signature.....

time, starting from the right, it can be rewritten in a form that requires at most one version of the multiplicand to be added to the partial product for each pair of multiplier bits.

Eg: multiplier bit-pair recoding



multiplier bit-pair recording table.

multiplier bit pair		multiplier bit on the right	multiplicand selected at position i
i+1	i	i-1	
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

Teacher's Signature

multiplication:

Example :

multiplicand = +13 (0 1 1 0 1)

multiplier = -6 (1 1 0 1 0)

```
  0 1 1 0 1
x 1 1 0 1 0
```

↓

1. 2's Complement of multiplicand

0 1 1 0 1

↓ 2's complement = 1's complement +1

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

2. multiplier bit pair recoding:

1 1 1 0 1 0 0

↓

0 0 −1 +1 −1 0

↓ ↓ ↓

0 −1 −2

3) multiplicand x multiplier bit pair recoding

```
          0   1   1   0   1
      x   0       -1      -2
    ---------------------------
          1   0   0   1   1   0
        1   0   0   1   1
  +   0   0   0   0   0
    ---------------------------
  (1) 1   1   0   1   1   0   0   1   0      (-78)
```

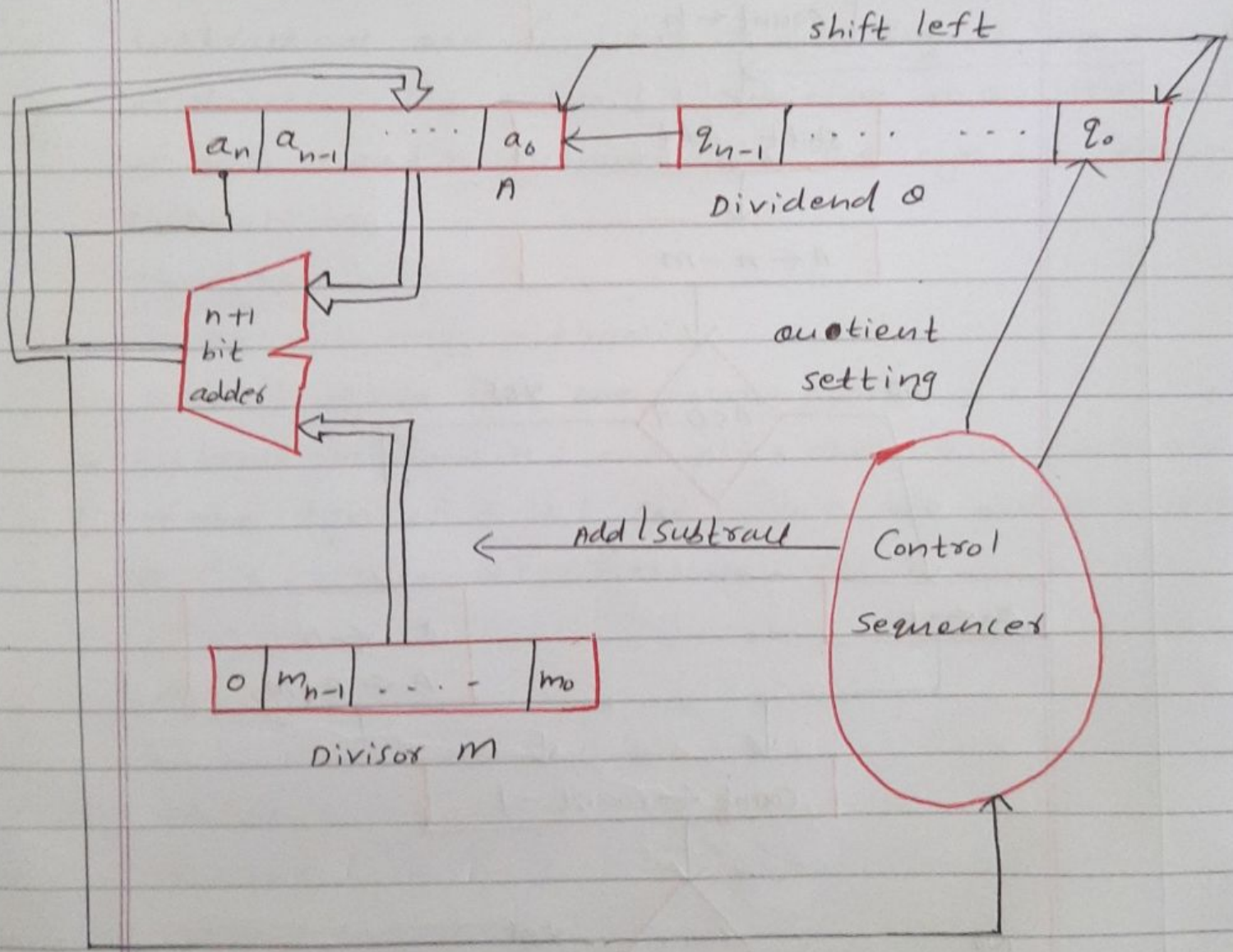
Here multiplication requiring only n/2 summands.

Teacher's Signature.....

Division restoring and non restoring Algorithm:

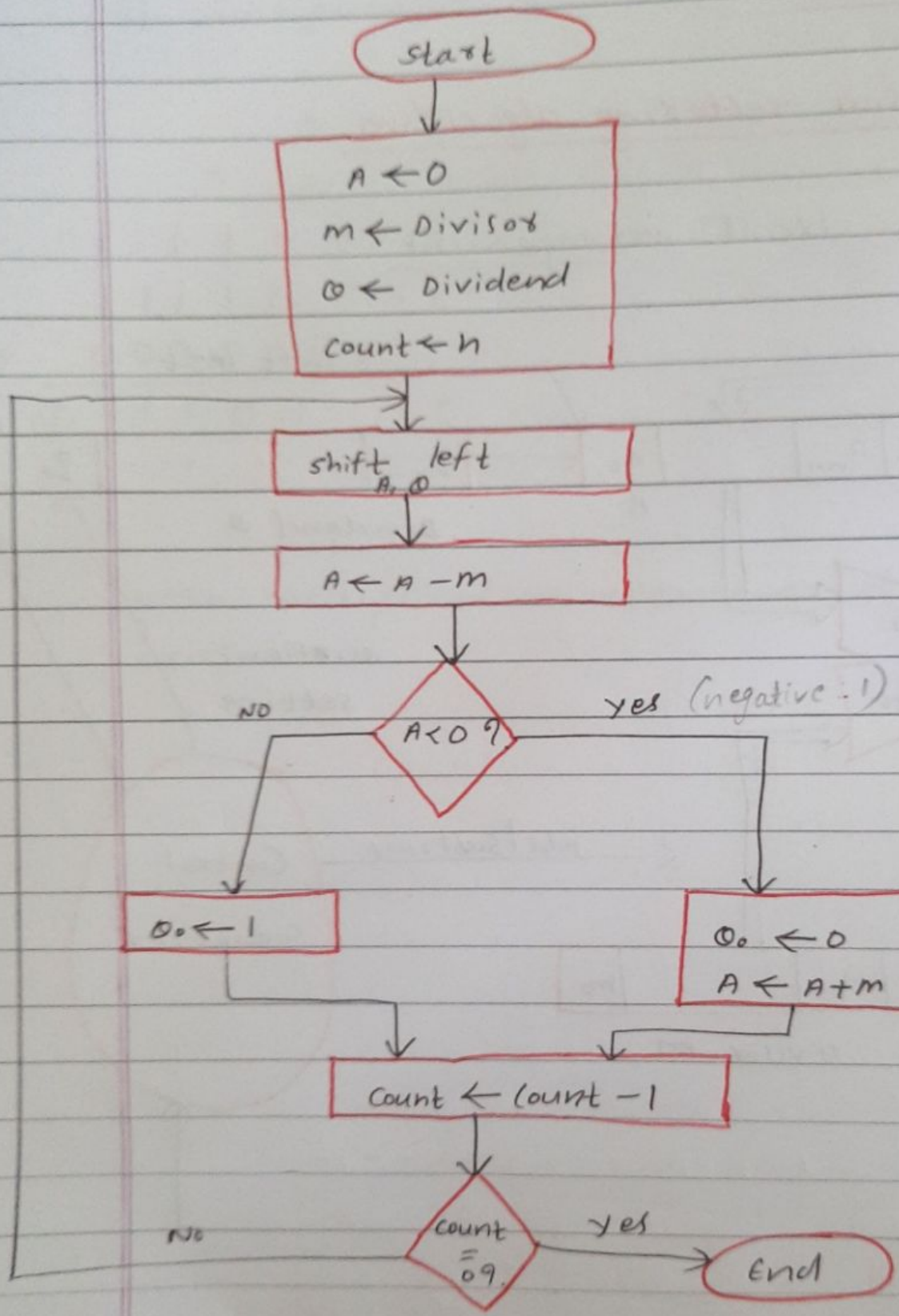
. Division restoring algorithm:

Logic circuit arrangement:



DAY:
DATE: / /

Algorithm:



Quotient in Q
Remainder in A.

Teacher's Signature.....

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 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.

Algorithm:

Do the following n times:

1. Shift A and Q left one binary position.
2. Subtract M from A, and place the answer back in A.
3. If the sign of A is 1, set q_0 to 0 and add M back to A (ie, restore A); otherwise, set q_0 to 1.

Example:

Divident = 1000

Divisor = 11

Normal Division:

```
    010  → quotient
11 | 1000
     00
     ---
     100
      11
     ---
     0010
       00
     ----
       10  → remainder
```

Teacher's Signature.....

A B C

DAY:

DATE: / /

Restoring division:

Divident = 1000

Divisor (M) = 1011 $\Rightarrow$ m' $\Rightarrow$ 0100 + 1 $\Rightarrow$ 01101

Count = 4

Operation	A	Q
Initially	00000	1000
shift left	00001	000 □
A ← A − M	+ 11101	
	11110	
Set Q_0	(1)	
A ← A + M	+ 11	
	00001	000 0

First cycle

shift left	00010	00 0 □
A ← A − M	+ 11101	
	11111	
Set Q_0	(1)	
A ← A + M	+ 11	
	00010	00 0 0

Second cycle

shift left	00100	0 0 0 □
A ← A − M	11101	
	00001	
Set Q_0	(0)	
	00001	0 0 0 1

Third cycle

Teacher's Signature.....

A B C

DAY :

DATE : / /

	A	Q
shift left	0 0 0 1 0	0 0 1
A ← A - M	1 1 1 0 1	
Set Q0.	1 1 1 1 1	
	1 1	
A ← A + M	0 0 0 1 0	0 0 1 0

7th cycle

Remainder

quotient

• Division nonrestoring algorithm:

The restoring - division algorithm can be improved by avoiding the need for restoring A after an unsuccessful subtraction. Subtraction is said to be unsuccessful if the result is negative.

Algorithm:

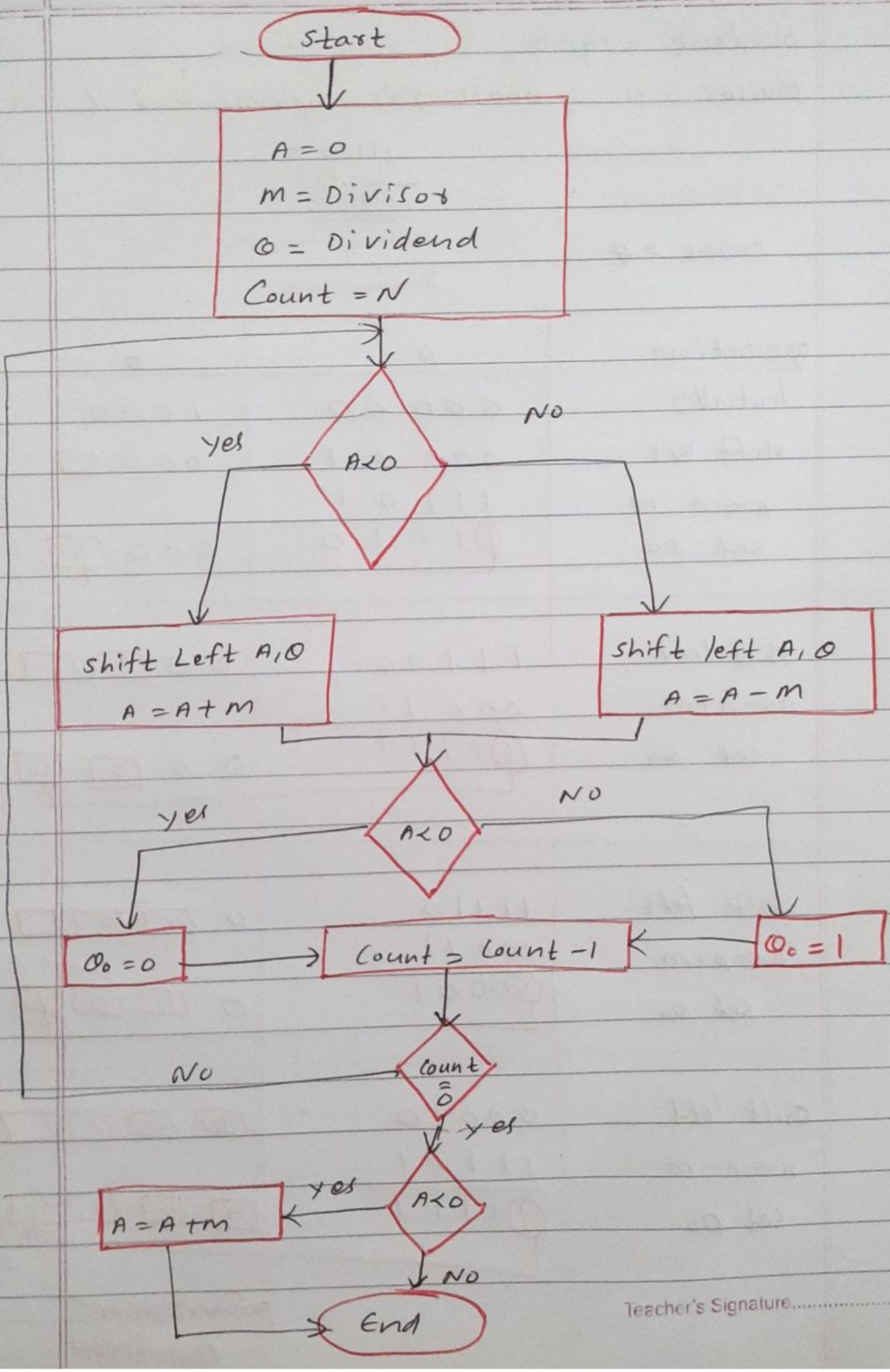
step 1: Do the following n times:

1.1. If the sign of A is 0, shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A.

1.2 Now, if the sign of A is 0, set q_0 to 1; otherwise, set q_0 to 0.

step 2: If the sign of A is 1, add M to A.

step 2 is needed to leave the proper positive remainder in A at the end of the n cycles of step 1.



Teacher's Signature.....

A B C

DAY:

DATE: / /

Example:

Divident = 1000

Divisor = 11 ⇒ 00011 ⇒ 2's complement of divisor

$$\begin{array}{r} 11100 + \\ 1 \\ \hline 11101 \end{array}$$

Count = 4

operation	A	Q	
Initially	0 0 0 0	1 0 0 0	
Shift left	0 0 0 0 1	0 0 0 □	
A ← A − M	1 1 1 0 1		1st cycle
set Q0	1 1 1 1 0	0 0 0 0	
Shift left	1 1 1 0 0	0 0 0 □	
A ← A + M	0 0 0 1 1		2nd cycle
set Q0	1 1 1 1 1	0 0 0 0	
Shift left	1 1 1 1 0	0 0 0 □	
A ← A + M	0 0 0 1 1		3rd cycle
set Q0	0 0 0 0 1	0 0 0 1	
Shift left	0 0 0 1 0	0 0 1 □	
A ← A − M	1 1 1 0 1		4th cycle
set Q0	1 1 1 1 1	0 0 1 0	

Teacher's Signature.....

Quotient.

A B C

DAY :

DATE : / /

$A = A + m$

$= 11111 +$

00011

00010

Restore remainder

Remainder

2. Floating point representation:

→ Floating point.

Numbers with fractions is called floating point numbers.

eg: 3.14_{10} (π)

2.718 (e)

• scientific notation:

It is a way of expressing numbers that are too big or too small to be conveniently written in decimal form.

$$\text{significand} \times 10^{\text{Exponent}}.$$

A number in scientific notation that has no leading 0's is called a normalized number.

eg: 1.0×10^{-9} is in normalized scientific notation

• binary number in scientific notation:

$$\text{significand} \times 2^{\text{Exponent}}.$$

In scientific notation, single digit to the left of the decimal point.

Computer arithmetic that support such numbers (binary point) is called floating point because it represents numbers in which the binary point is not fixed, as it is for integers. The programming language C uses the name float for such numbers.

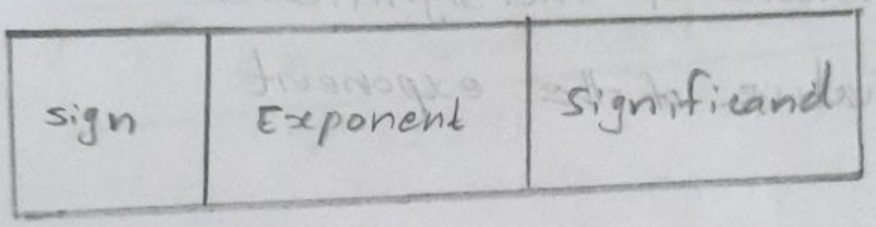
In scientific notation, numbers are represented as a single nonzero digit to the left of the binary point. In binary, the form is

$$1.xxxxxxxx_2 \times 2^{yyyy}.$$

• Floating point representation:

1. MIPS representation:

• sign and magnitude form:



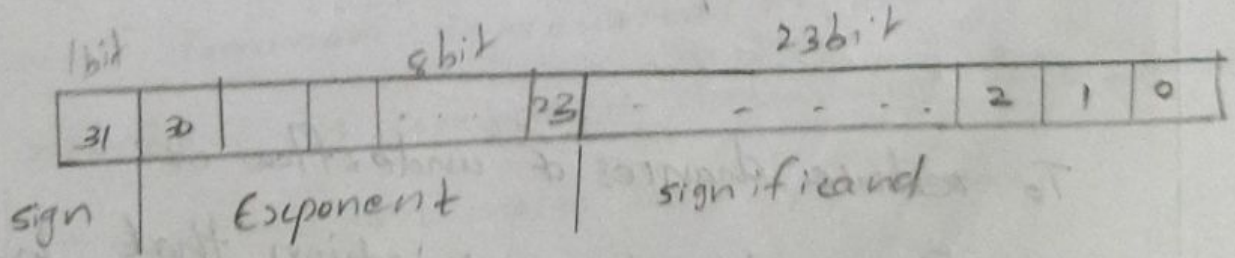
where

s = sign of the floating point number (1 bit)

Exponent = value of the 8 bit exponent field. (including the sign of the exponent)

significand = 23 bit number in the fraction.

This representation is called sign and magnitude.



In general, floating point numbers are of the form:

$$(-1)^{S} \times F \times 2^{E}$$

where

S ⇒ sign (value)

F → value of the significand

E → value of the exponent

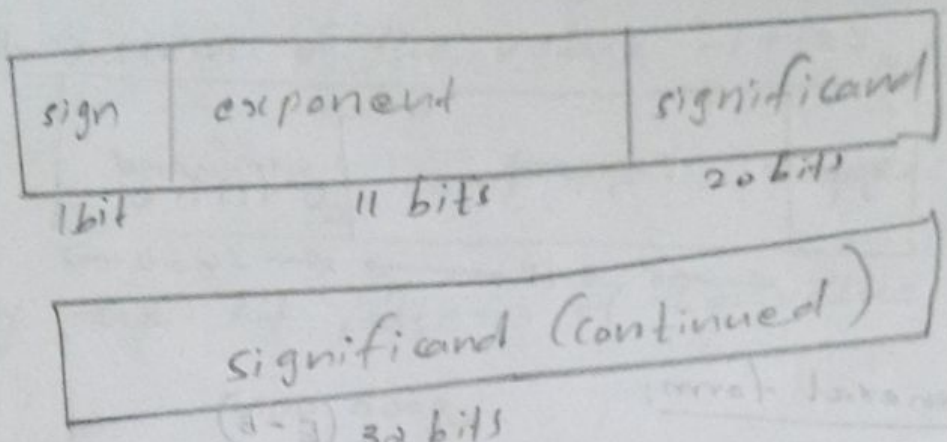
Range:

(small) $2.0_{10} \times 10^{-38}$ To $2.0_{10} \times 10^{38}$ (large)

- overflow: the exponent is too large to be represent in the exponent field.
- underflow: Negative exponent is too large to fit in the exponent field.

To reduce chances of underflow or overflow is to use a notation that has a larger exponent. In C this is called double and operations on doubles are called double precision floating point arithmetic.

. Double floating point representation:



Range:

$2.0_{10} \times 10^{-308}$ to $2.0_{10} \times 10^{308}$

advantage

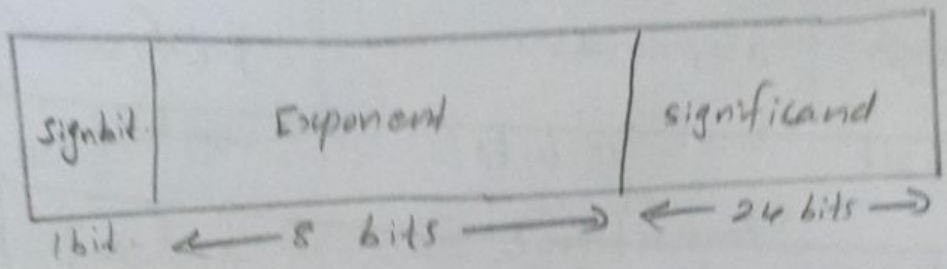
- greater accuracy (large significand).

2. IEEE 754 floating point standard.

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.

. single precision Format:



general form:

$$(-1)^S \times (1 + \text{significand}) \times 2^{(E-B)}$$

where

E → value in the exponent field.

significand → fraction between 0 and 1.

B → Bias

The bits of the significand from left to right $s_1, s_2, s_3, \ldots$ then the value is:

$$(-1)^S \times \left(1 + (s_1 \times 2^{-1}) + (s_2 \times 2^{-2}) + \cdots\right) \times 2^E$$

. biased notation:

It is a way of storing a range of values that doesn't start with zero.

$\text{Bias} = 2^{k-1} - 1$ where k is the no. of bits in the binary exponent.

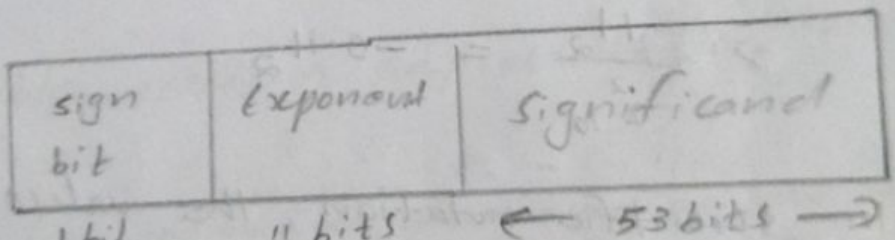
IEEE 754 uses a bias of 127 for single precision:, so −1 is represented by the bit pattern of the value $-1+127_{10}$ or 126_{10}

$= 01111110_2$ and +1 is represented by the bit pattern of the value $1+127_{10}=128$

$= 1000\ 0000_2$.

ie $(-1)^S \times (1+\text{significand}) \times 2^{(E-127)}$

- Double precision format:



general form:

$(-1)^S \times (1+\text{significand}) \times 2^{E-1023}$

eg: show the IEEE754 binary representation of the number -0.75_{10} in single and double precision:

solution:

the number -0.75_{10} is also

$$-\frac{3}{4}_{10} \quad \text{or} \quad \frac{-3}{2^2}_{10}$$

It is also represented by the binary fraction:

$$\frac{-11_2}{2^2_{10}} = -0.11_2$$

00 - 0
01 - 1
10 - 2
11 - 3

In scientific notation, the value is

$$-0.11_2 \times 2^0$$

and in normalized scientific notation, it is

$$-1.1_2 \times 2^{-1}$$

The general representation for a single precision number is:

$$(-1)^{S} \times (1 + \text{significand}) \times 2^{(\text{Exponent} - 127)}$$

$$(-1)^{1} \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000_{2}) \times 2^{(126-127)}$$

The single precision binary representation of -0.75_{10} is then.

1	8	24
1	01111110	1000 0000 0000 0000 0000 0000

$2^7 = 128$
$2^6 = 64$
$2^5 = 32$
$2^4 = 16$
$2^3 = 8$
$2^2 = 4$
$2^1 = 2$
$2^0 = 1$

The double precision representation is:

$$(-1)^{1} \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{2}) \times 2^{(1022-1023)}$$

1	01111111110	1000 0000 0000 0000 0000
1	11 bit.	20 bits

0000 0000 0000 0000 0000 0000 0000 0000

32 bits

q2: Converting Binary to decimal Floating point.

31	30 ... 23	22 ... 0
1	1000 0001	0100 0000 0000 0000 0000 0000....

solution

general form:

$$(-1)^S \times (1 + \text{significand}) \times 2^{(\text{Exponent} - \text{Bias})}$$

where

$S \Rightarrow 1$

$E \Rightarrow 2^0 + 2^7 \Rightarrow 1 + 128 \Rightarrow 129$

Bias $\Rightarrow 127$

significand $\Rightarrow 2^{-2} \times 1 = \frac{1}{2^2} = \frac{1}{4} = 0.25$

$$= (-1)^1 \times (1 + 0.25) \times 2^{129-127}$$

$$= -1 \times 1.25 \times 2^2$$

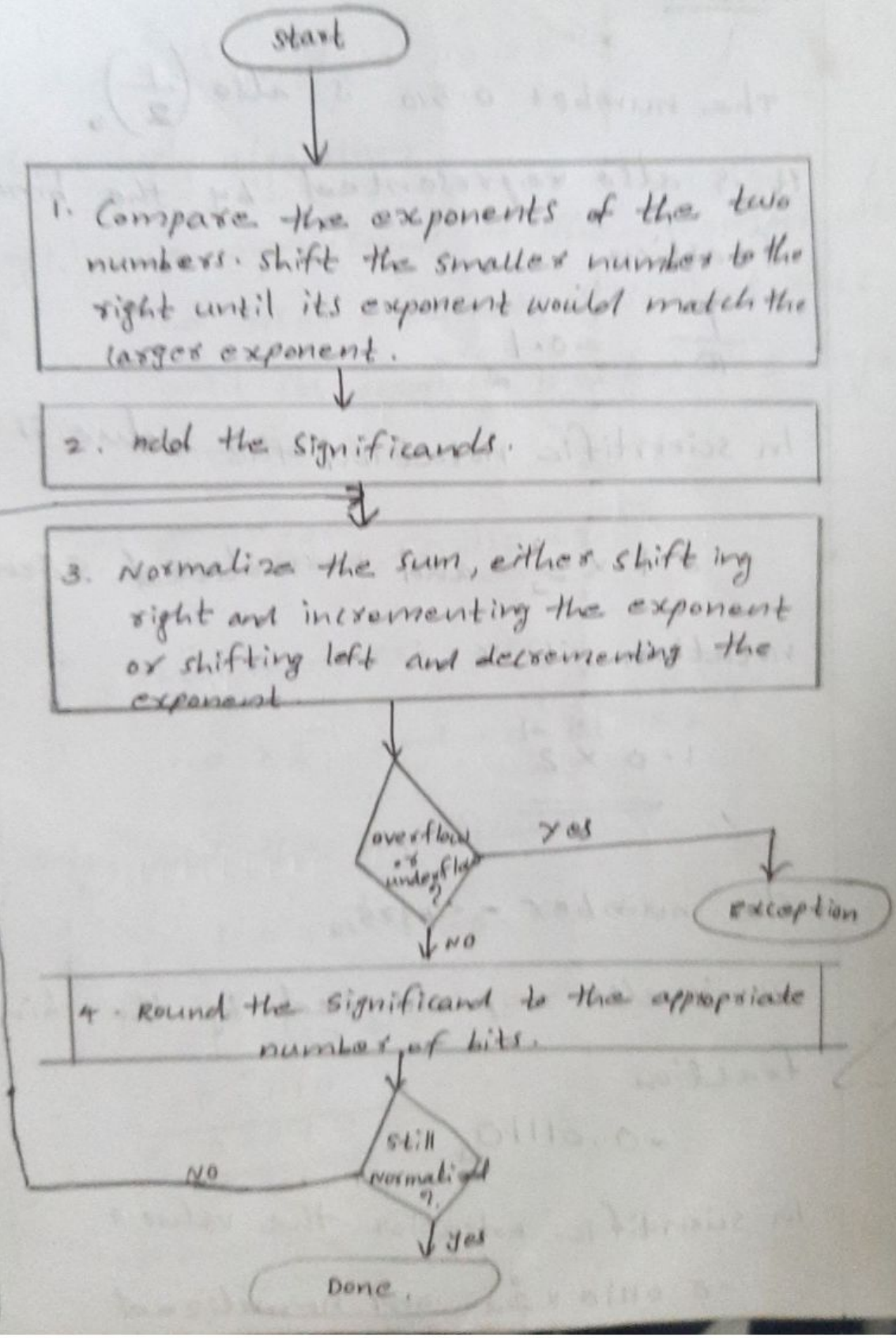
$$= -1 \times 5$$

$$= -5.0$$

Floating point operation:

1. Floating-point addition:

Algorithm:



eg: adding the numbers 0.5_{10} and $-.4375_{10}$ in binary using floating point addition algorithm.

solution:

The number 0.5_{10} is also $\left(\frac{1}{2}\right)_{10}$

It is also represented by the binary fraction.

$$\frac{1}{10} = 0.1_2$$

In scientific notation the value is

$0.1 \times 2_2^0$ and normalized scientific notation, it is

$$1.0 \times 2^{-1}$$

The number $-.4375_{10}$

It is also represented by the binary fraction

$$-0.01110_2$$

In scientific notation the value is

$-0.01110 \times 2_2^0$ and normalized

scientific notation it is:

-1.110×2^{-2}

so the binary numbers are

1.0×2^{-1} and -1.110×2^{-2}.

Then apply algorithm:

step1: the significand of the number with the lesser exponent is shifted right until its exponent matches the larger number:

$-1.110 \times 2^{-2} = -0.1110 \times 2^{-1}$

Then numbers are:

1.0×2^{-1} and -0.1110×2^{-1}

step2: add the significands:

$1.0 \times 2^{-1} + -0.1110 \times 2^{-1}$

$$\begin{array}{r} {\scriptstyle 1\ 1\ 1\ 1\ 0} \\ 1.0000 + \\ -0.1110 \\ \hline 0.0010 \end{array}$$

$= 0.0010 \times 2^{-1}$

addition

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

subtraction

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

step3: Normalize the sum, checking for overflow or underflow.

$$0.0010 \times 2^{-1} \Rightarrow 1.000 \times 2^{-4}$$

$$\Rightarrow -126 \leq -4 \leq 127$$

there is no overflow or underflow.

step4: Round the sum.

$$1.000 \times 2^{-4}$$

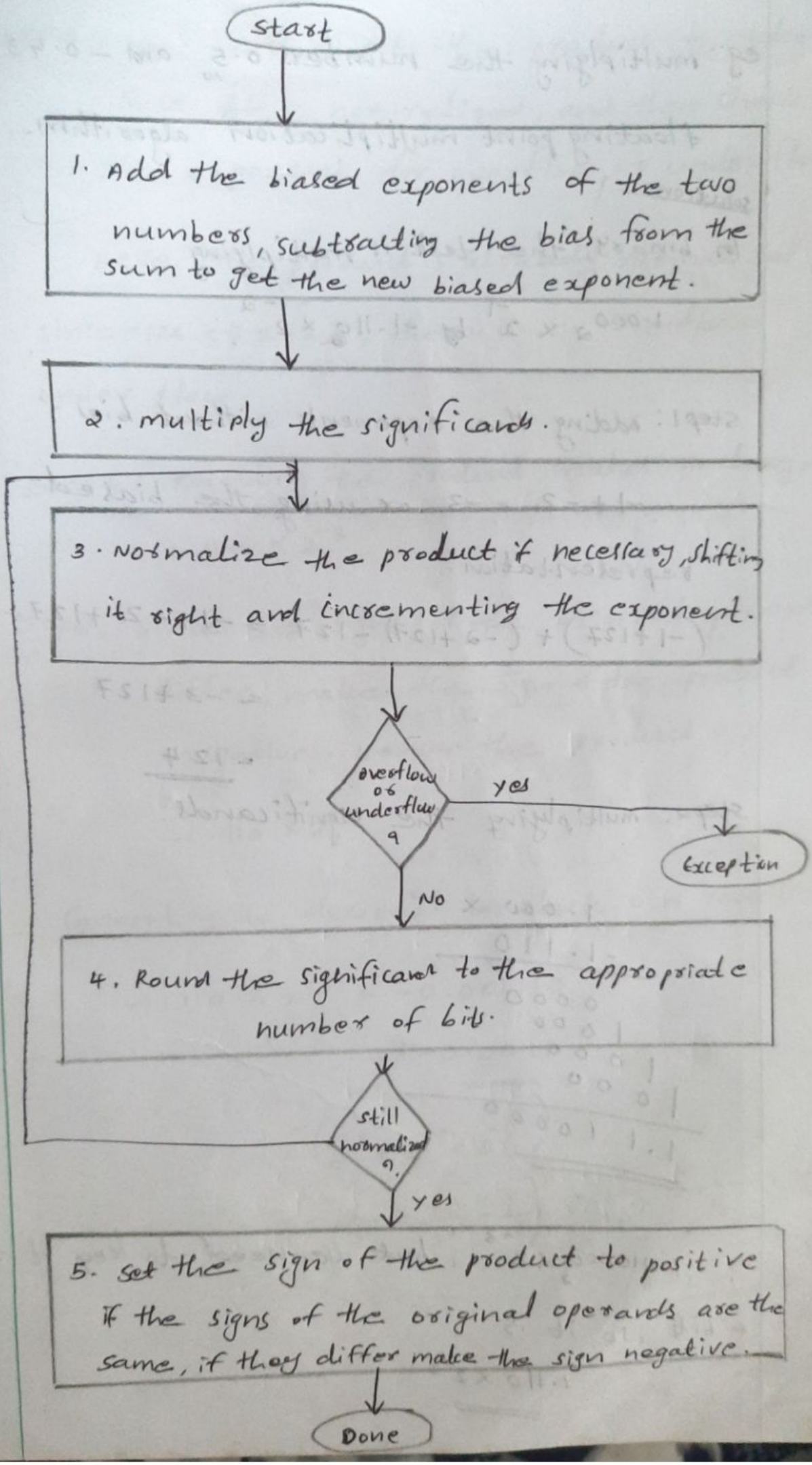
this sum is then.

$$1.000 \times 2^{-4} = 0.0001_2$$

$$= \frac{1}{24_{10}} = \frac{1}{16_{10}} = 0.0625_{10}$$

2. Floating point multiplication.

Algorithm:


start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent.
2. multiply the significands.
3. Normalize the product if necessary, shifting it right and incrementing the exponent.
overflow or underflow ?
yes
Exception
No
4. Round the significand to the appropriate number of bits.
still normalized ?
yes
5. Set the sign of the product to positive if the signs of the original operands are the same, if they differ make the sign negative.
Done

eg: multiplying the numbers 0.5_{10} and -0.4375_{10} using floating point multiplication algorithm.

solution:

In binary, the task is multiplying

$1.000_2 \times 2^{-1}$ by $-1.110_2 \times 2^{-2}$.

step1: adding the exponents without bias:

$-1 + -2 = -3$ or using the biased representation:

$(-1+127) + (-2+127) - 127 = -1 + -2 + 127 + 127 - 127$

$= -3 + 127$

$= 124$

step2: multiplying the significands:

$$\begin{array}{r} 1.000 \times \\ 1.110 \\ \hline 0000 \\ 1000 \\ 1000 \\ 1000 \\ \hline 1.1\ 10000 \end{array}$$

$1.110000_2 \times 2^{-3}$, but we need to keep it to 4 bits, so it is

1.110×2^{-3}

step3: Now, we check the product to make sure it is normalized, and then check the exponent for overflow or underflow.

Here the product is already normalized and since $-126 \leq -3 \leq 127$, there is no overflow or underflow.

step4: Rounding the product makes no change.

1.110×2^{-3}.

step5: since the signs of the original operands differ, make the sign of the product negative. Hence the product is.

-1.110×2^{-3}

Converting to decimal to check our results.

$-1.110 \times 2^{-3} = -0.001110_2$

$= -0.00111$

$= -(2^{-3} + 2^{-4} + 2^{-5})$

$= -\left(\frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^5}\right) = -\left(\frac{1}{8} + \frac{1}{16} + \frac{1}{32}\right)$

$= -\left(\frac{4+2+1}{32}\right)$

$= \frac{-7}{32} = -0.21875$

0. Adding the numbers $9.999_{10} \times 10^{1} + 1.610_{10} \times 10^{1}$ using the algorithm 9.

step1: $1.610_{10} \times 10^{-1} = 0.1610_{10} \times 10^{0}$

$= 0.01610_{10} \times 10^{1}$

Step2: addition of significands:

$$\begin{array}{r} 9.99900 + \\ 0.01610 \\ \hline 10.01510 \end{array}$$

ie 10.01510×10^{1}

$10.0151_{0} \times 10^{1}$

Step3: normalized form

$1.0015_{10} \times 10^{2}$

step4 = $1.002_{10} \times 10^{2}$

02. multiply the numbers $1.110_{10} \times 10^{10} \times 9.200_{10} \times 10^{-5}$

step 1:

New exponent $= 10 + -5 = 5$

New exponent in bias:

$= (10 + 127) + (-5 + 127) - 127$

$= 137 + 122 - 127$

$= 259 - 127$

$= 132$ $(127+5)$

Step 2: multiplication of the significands.

```
    9.200 x
    1.110
  ---------
    0000
   9200
  9200
 9200
---------
10.212000
```

ie 10.212000_{10}

$10.212000_{10} \times 10^{5}$

step 3 : normalized form

$$1.0212000_{10} \times 10^6$$

step 4 : $1.021_{10} \times 10^6$

step 5 : sign of the product

$$+1.021_{10} \times 10^6$$