5. Arithmetic Logic Unit

⇒ Data can be represented in 2 forms -
1) fixed point
2) floating point.

⇒ Arithmetic instruction is used to manipulate the data.

⇒ negative fixed point numbers are mostly represented in 2's complement form.

⇒ negative floating point numbers are represented in signed magnitude form.

Addition Algorithm

⇒ For addition, if the sign of two numbers are same, then add the two numbers & keep the sign of A

⇒

(+A) + (+B)	+(A+B)
(-A) + (-B)	-(A+B)

⇒ If operation is addition & the signs of two numbers are different then compare the magnitudes, subtract the smaller one from bigger one & keep the sign of A of to the result otherwise complement A

	A>B	A<B	A=B
(+A) + (-B)	+(A-B)	-(B-A)	+(A-B)
(-A) + (+B)	-(A-B)	+(B-A)	+(A-B)

Subtraction Algorithm

	A>B	B>A	A=B
⇒ (+A) - (+B)	+(A-B)	+(B-A)	+(A-B)

If operation is subtraction & operands are having same sign then compare the magnitude of A & B & keep the A's sign to result otherwise A's complement

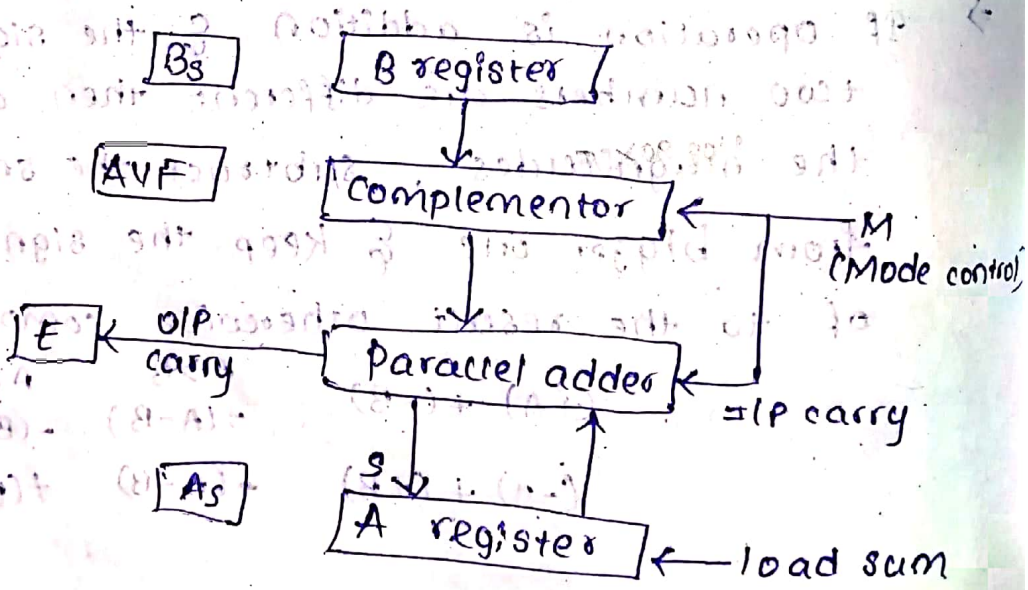
(-A) - (-B)	-(A-B)	+(B-A)	+(A-B)

⇒ If operation is subtraction & operands are having different signs then add two operands & keep the A's sign to the result.

(+A) - (-B) → +(A+B)

(-A) - (+B) → -(A+B)

Hardware Implementation of addition & subtract



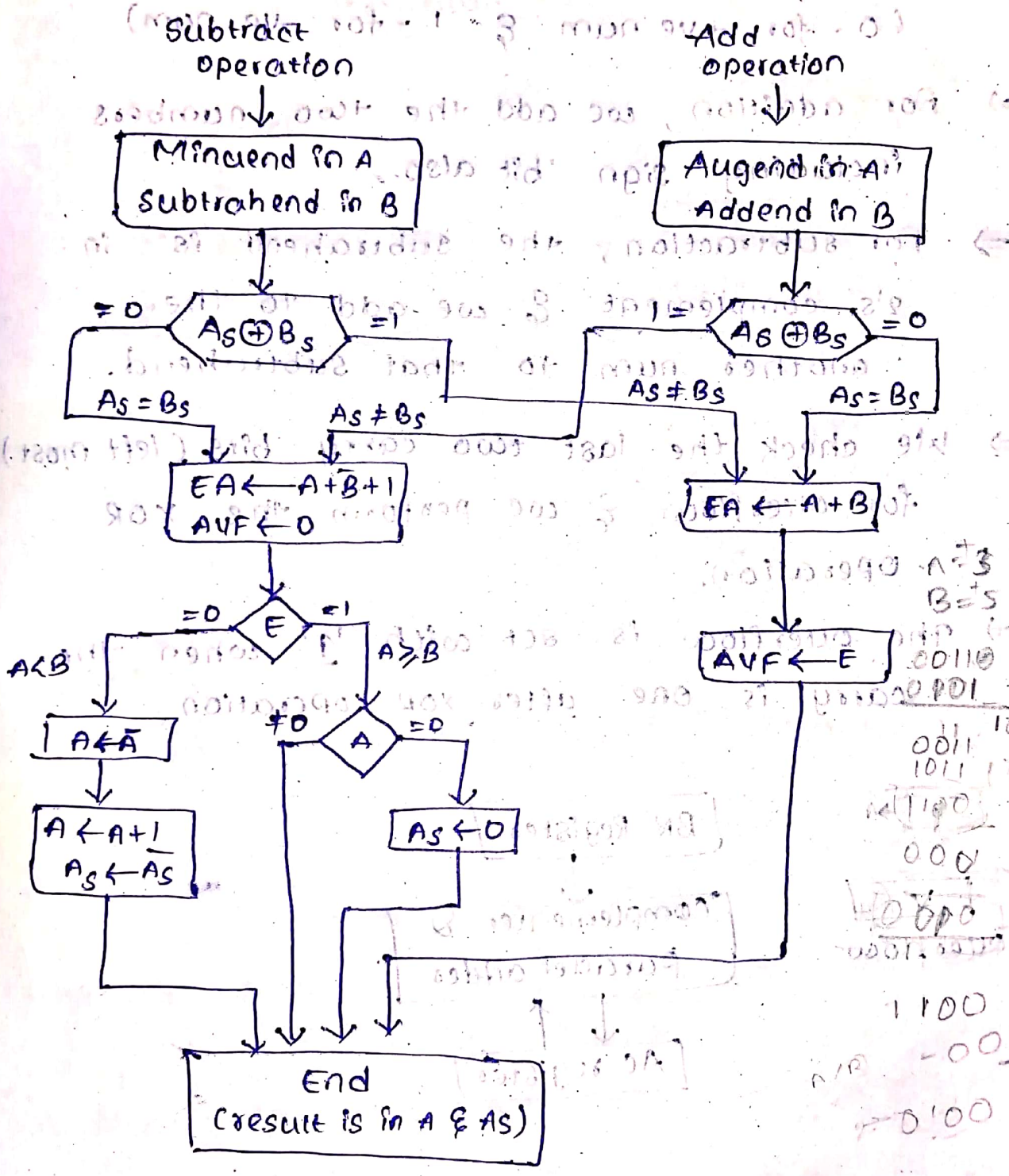
⇒ Bs, AVF, E, As are all flip flops

⇒ Complementor provides B o/p & also 1's & complement of B to the parallel adder based on Mode

If M = 0, B value ⟶ parallel adder

If M = 1, complement of B + I/P carry (1) ⟶ parallel adder

Flowchart: For add & subtract operations



Hardware for signed 2's complement addition and subtraction:

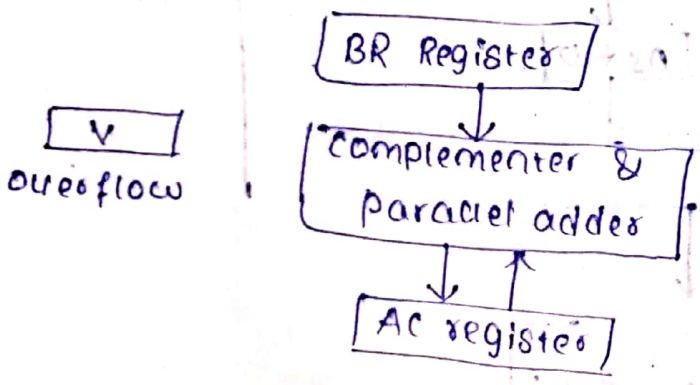
⇒ To represent signed binary numbers, we use left most bit.
(0 - for +ve num & 1 - for -ve num)

⇒ For addition, we add the two numbers including sign bit also.

⇒ For subtraction, the subtrahend is in 2's complement & we add to the another num to that subtrahend.

⇒ We check the last two carry bits (left most) for overflow & we perform the XOR operation.

⇒ The overflow is set with '1' when the carry is one after XOR operation.

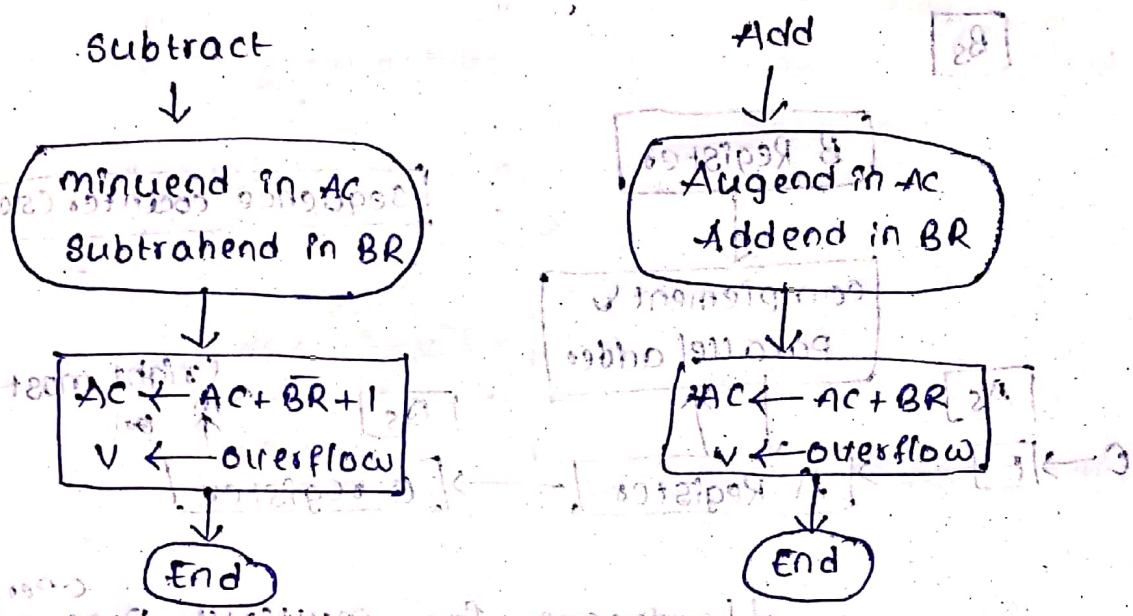


⇒ In accumulator register we take the num A with signed bit & BR register also. we take the num B with sign bit.

⇒ After that we add the A & B num in & finally store in AC register and it checks overflow also.

⇒ If we perform subtraction, it calculates 2's complement for B subtrahend and add with A num or minuend & store in AC register.

Algorithm for sign 2's complement for addition & subtraction :



Multiplication Algorithm

The first one is multiplicand & 2nd one is multiplier

⇒ If the bit is 1 in multiplier, the partial product is same as multiplicand.

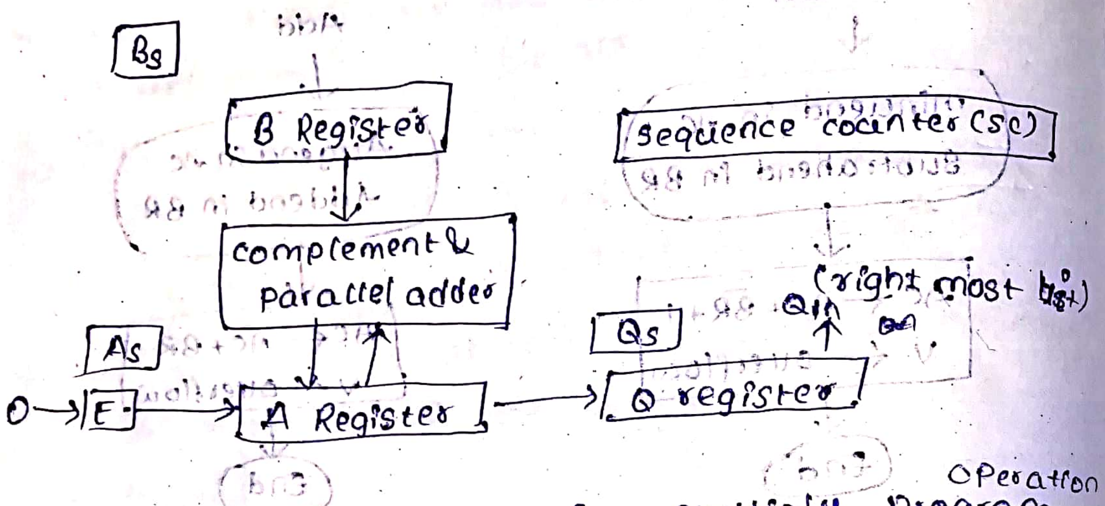
⇒ If the bit is 0 in multiplier, the partial product is all are zeroes.

⇒ finally we add the partial products.

To implement multiplication in hardware the following are needed :

* We need to use the adder for adding of two binary numbers & accumulate the partial product in register.

* Instead of shifting the multiplicant to the left, we shift to the right. (partial product) with '0' &
* When we need not to multiply when we need to add the partial product when the corresponding multiplier is 0.



Operation
Hardware for multiply Program

⇒ The multiplier value is stored in Q register and multiplicand is in B register, the partial product is in A register.
(Initially A = 00000)

⇒ sequence counter contains the number which is equal to the no. of bits in B multiplier.

10111
× 10011
10111
10111
1 00010

E: 0 A: 10111 Q: 10011

Shift one bit to the right. (after adding A) with B)

⇒ 0 01011 11001

⇒ After shifting decrement SC.

Scanned with CamScanner

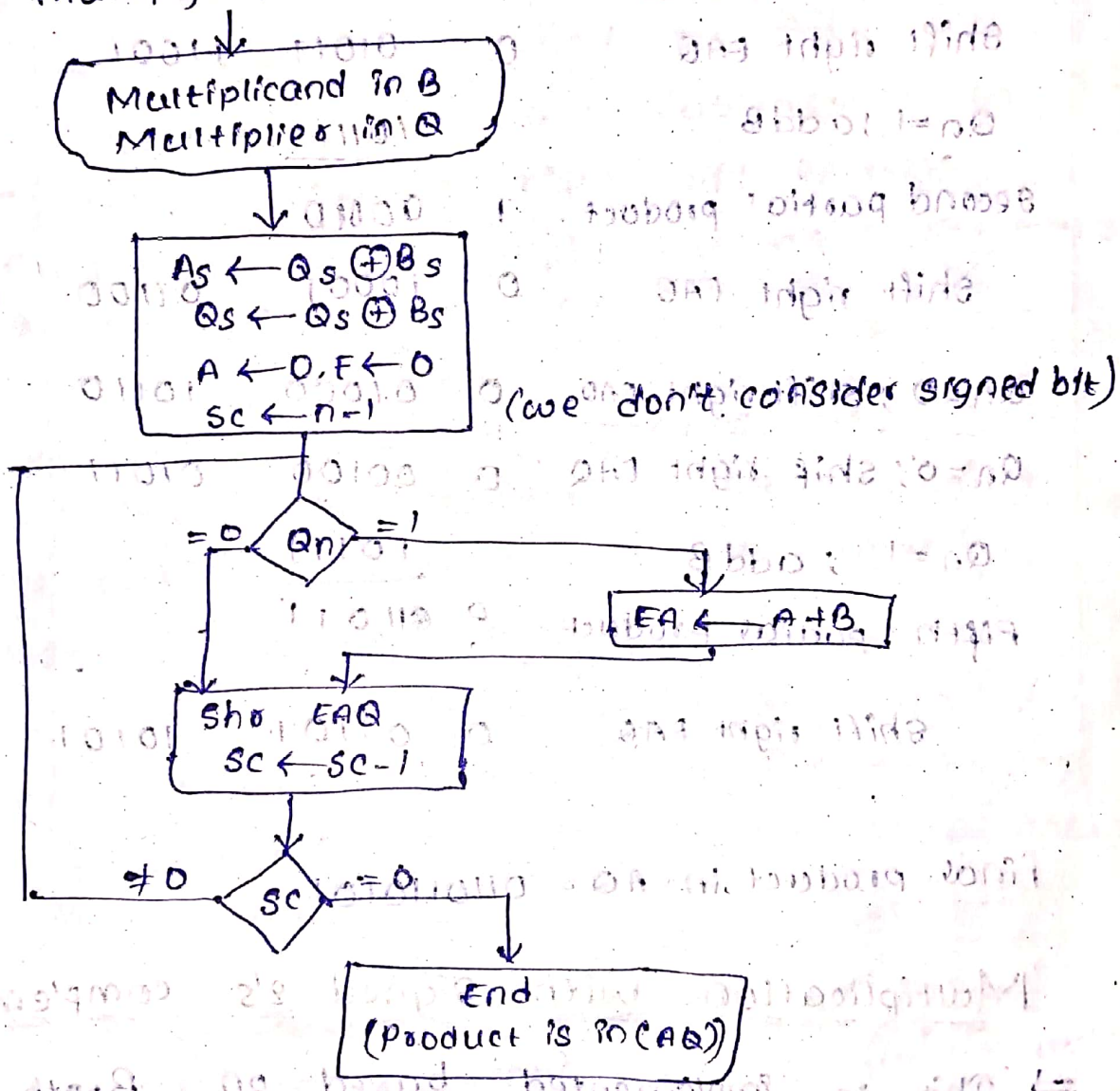
⇒ 10111
01011
+ 00010
1 ⇒ 00010

E	A	Q
1	00010	1001 1

Shift to right

⇒	0	10001	01100 → Qn

multiply Operation



Numerical example for binary Multiplier :-

Multiplicand B=10111	E	A	Q	Sc
Multiplier in Q	0	00000	10011	101
$Q_n = 1$; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
$Q_n = 1$; add B		10111		
second partial product	1	00010		
shift right EAQ	0	10001	01100	011
$Q_n = 0$; shift right EAQ	0	01000	10110	010
$Q_n = 0$; shif right EAQ	0	00100	01011	001
$Q_n = 1$; add B		10111		
Fifth partial product	0	11011		
shift right EAQ	0	01101	10101	000

Final product in AQ = 0110110101

Multiplication with signed 2's complement form

⇒ This is implemented based on Booth algorithm

⇒ Booth algorithm is a procedure for multiplication of 2 binary numbers.

$2^K - 2^m$

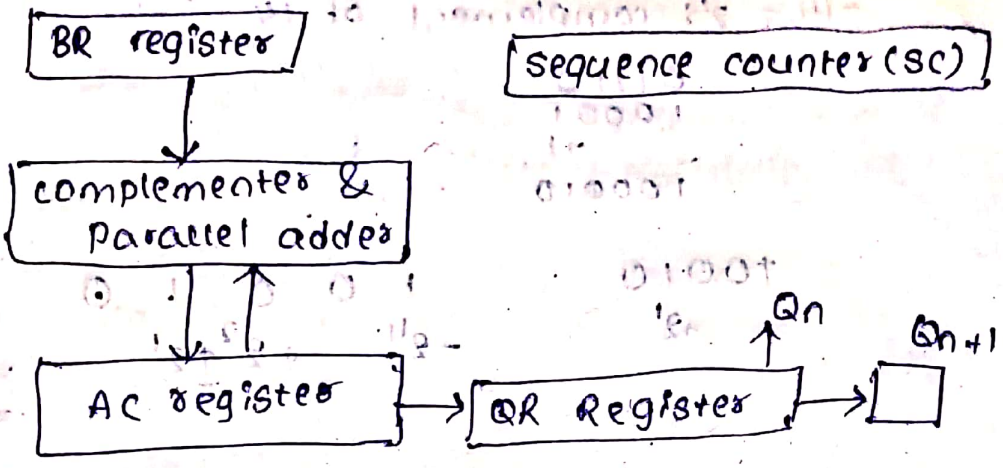
0 1 1 1 0 (multiplier)

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

$2^{K+1} - 2^m = 2^{3+1} - 2^1 = 16 - 2 = 14$

$M \times 01110 = M \times 14 = \underbrace{M \times 2^4}_{\text{shift 4 times}} - \underbrace{M \times 2^1}_{\text{shift 1 bit}}$

Hardware for Booth algorithm:



In BR, AC & QR registers, we store the value including the signed bit also

After performing addition & subtraction, we perform arithemetic right shift operations. (signed bit remain unchanged while performing arithem right shift operation)

⇒ prior to shifting the multiplicand may be added to partial product, subtracted from the partial product or left unchanged based on following rules:-

* The multiplicand is subtract from partial product when encountering the first least significant 1 in the strings of ones in the multiplier. ... 0110 (right ← left)

* The multiplicand is added to the partial product when encounters the first zero in a strings of zeroes in the multiplier

... 0110

* The partial product doesn't change when the multiplier bit is identical to the previous bit.

-14 = 2's complement of 14

```
  01110
  10001
     +1
 ------
  10010
```

10 - sub
01 - add

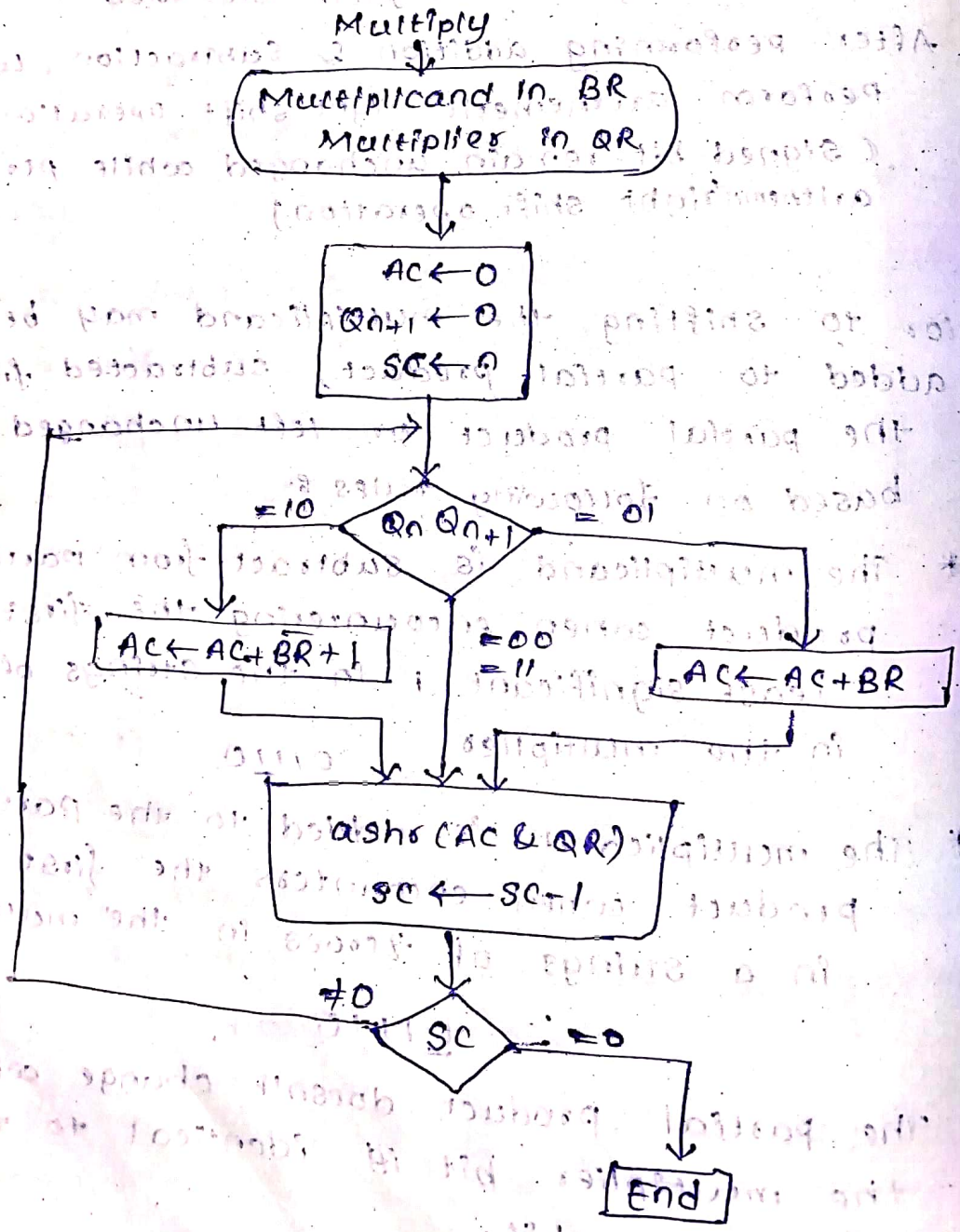
$+0010$

$+2^1$

$1\ 0\ 0\ 1\ 0$

$-2^4 + 2^2 + 2^1 = -16 + 4 - 2 = -14$

Algorithm



Example:

$(-9) \times (-13) = +117$

multiplicand BR = -9
multiplier QR = -13

$01001 \rightarrow 10110 \xrightarrow{+1} -9 = 10111$

$01101 \rightarrow 10010 \xrightarrow{+1} 10011 = -13$

BR = 10111
$\overline{BR}+1 = 01001$

Qn	Qn+1		AC	QR	Qn+1	SC
			00000	10011	0	101
1	0	Subtract BR	01001			
			01001			
		ashr	00100	11001	1	100
1	1	ashr	00010	01100	1	011
0	1	add BR	10111			
			11001	01100	1	
		ashr	11100	10110	0	010
0	0	ashr	11110	01011	0	001
1	0	subtract BR	01001			
			① 00111	01011	0	
		ashr	00011	10101	1	000

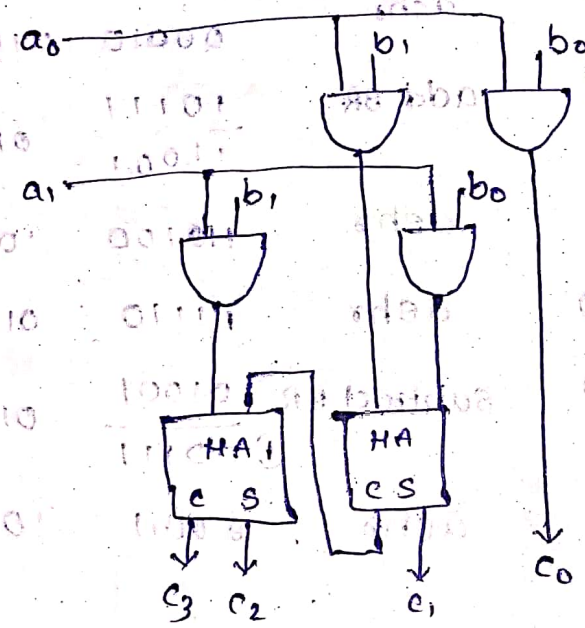
final product = 0001110101
= 117

Array Multiplier

* It is the fastest way for multiplying 2 binary numbers by using combinational circuit.
* It performs only one operation & gives the final result.
* Drawback :- not economical

multiplicand :- $b_1 b_0$ } 2-bit binary number
multiplier :- $a_1 a_0$

$$\begin{array}{r} b_1\ b_0 \\ a_1\ a_0 \\ \hline a_0b_1\ \ a_0b_0 \\ a_1b_1\ \ a_1b_0 \\ \hline c_3\ c_2\ c_1\ c_0 \end{array}$$



2 bit by 2 bit array multiplier

⇒ If three bits were there in either multiplier or multiplicand then we use full adder.

⇒ If more than 3 bits are there then we use parallel adder.

multiplicand - J bits
multiplier - K bits
then, and gates :- J x K
Parallel adder bits :- (J-1) x K

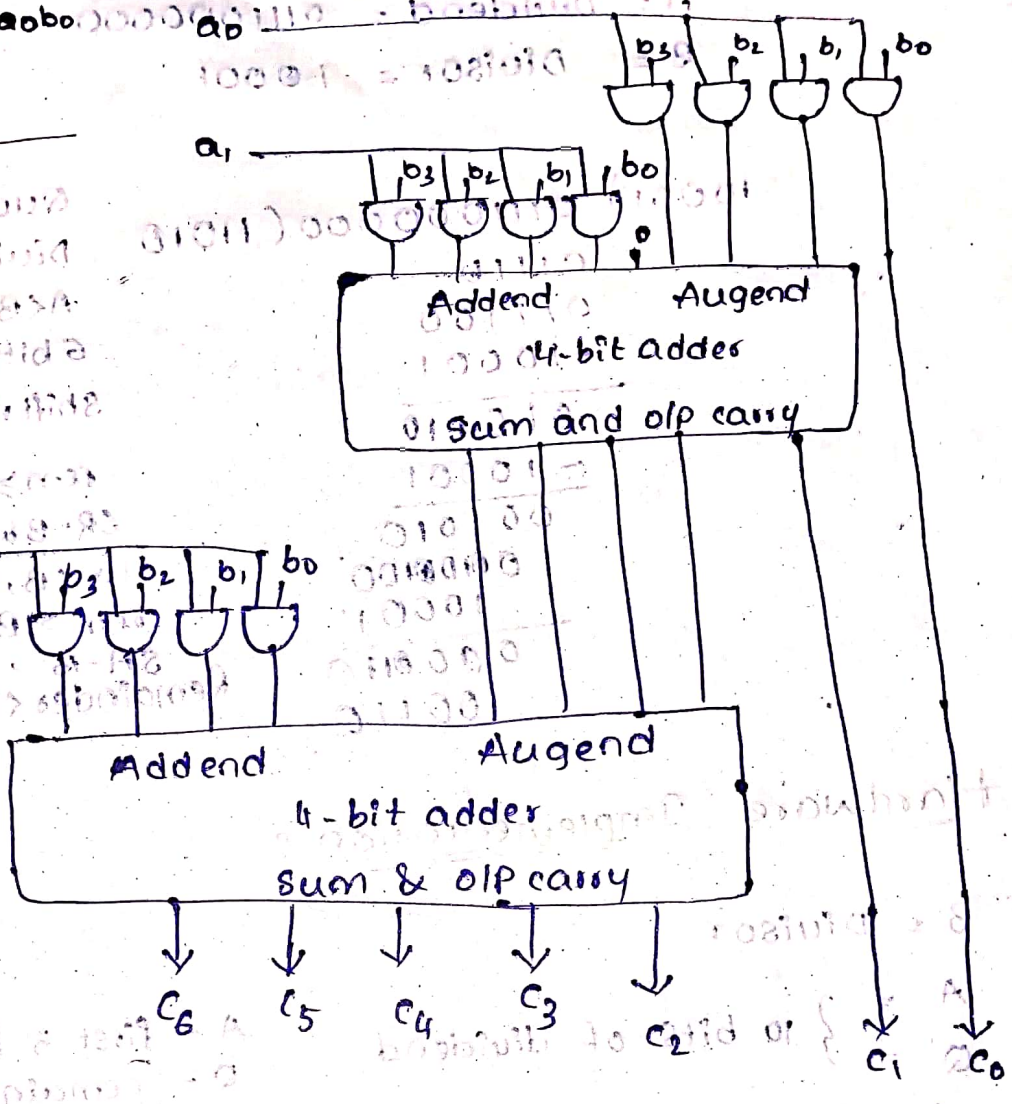
$b_3 b_2 b_1 b_0$

$\times\ a_2 a_1 a_0$

$a_0b_3\ a_0b_2\ a_0b_1\ a_0b_0$

$a_1b_3\ a_1b_2\ a_1b_1\ a_1b_0$

$a_2b_3\ a_2b_2\ a_2b_1\ a_2b_0$



no. of terms in result = J+K

no. of parallel adders = K-1

no. of levels = K

Division Algorithm

A = Dividend = 01110000000

B = Divisor = 10001

```
10001) 01110000000 (11010
       01110
       01110
      -10001
       -----
        01010
       -10001
       -------
        01000
         10001
         -----
         00110
```

Quotient = Q
Dividend = A
A < B
6 bits A ≥ B
Shift right B & Subtract, enter 1 in Q
rem ≥ B
SR-B & enter 1 in Q
rem < B, enter 0 in Q, SR B
rem ≥ B
SA-B, enter 1 in Q
Remainder < B; enter 0 in Q

Hardware Implementation :-

B = Divisor

A } 10 bits of dividend
Q }

A - first 5 bits
Q - remaining half bit

E = carry register (initially 0)

* Shift EAQ to the left
* Add 2's complement of B to the EAQ
* If E=1 then A ≥ B, set Q=1
  E=0, then A < B, restore A
* Decrement sequence counter

$B = 10001$, $\bar{B}+1 = 01111$

	E	A	Q	Sc
Dividend		01110	00000	5
shl EAQ	0	11100	00000	
add $\bar{B}+1$		01111		
	1	01011		
E=1, set $Q_n=1$	1	01011	00001	4
shl EAQ	0	10110	00010	
add $\bar{B}+1$		01111		
	1	00101		
E=1, set $Q_n=1$	1	00101	00011	3
shl EAQ	0	01010	00110	
add $\bar{B}+1$		01111		
	0	11001		
leave E=0, set $Q_n=0$ & add B		10001		
	1	01010	00110	2
shl EAQ	0	10100	01100	
add $\bar{B}+1$		01111		
	1	00011		
E=1, set $Q_n=1$	1	00011	01101	1
shl EAQ	0	00110	11010	
add $\bar{B}+1$		01111		
	0	10101		
E=0, leave Q_n & add B		10001		
	1	00110	11010	0

result = 00110 (remainder)

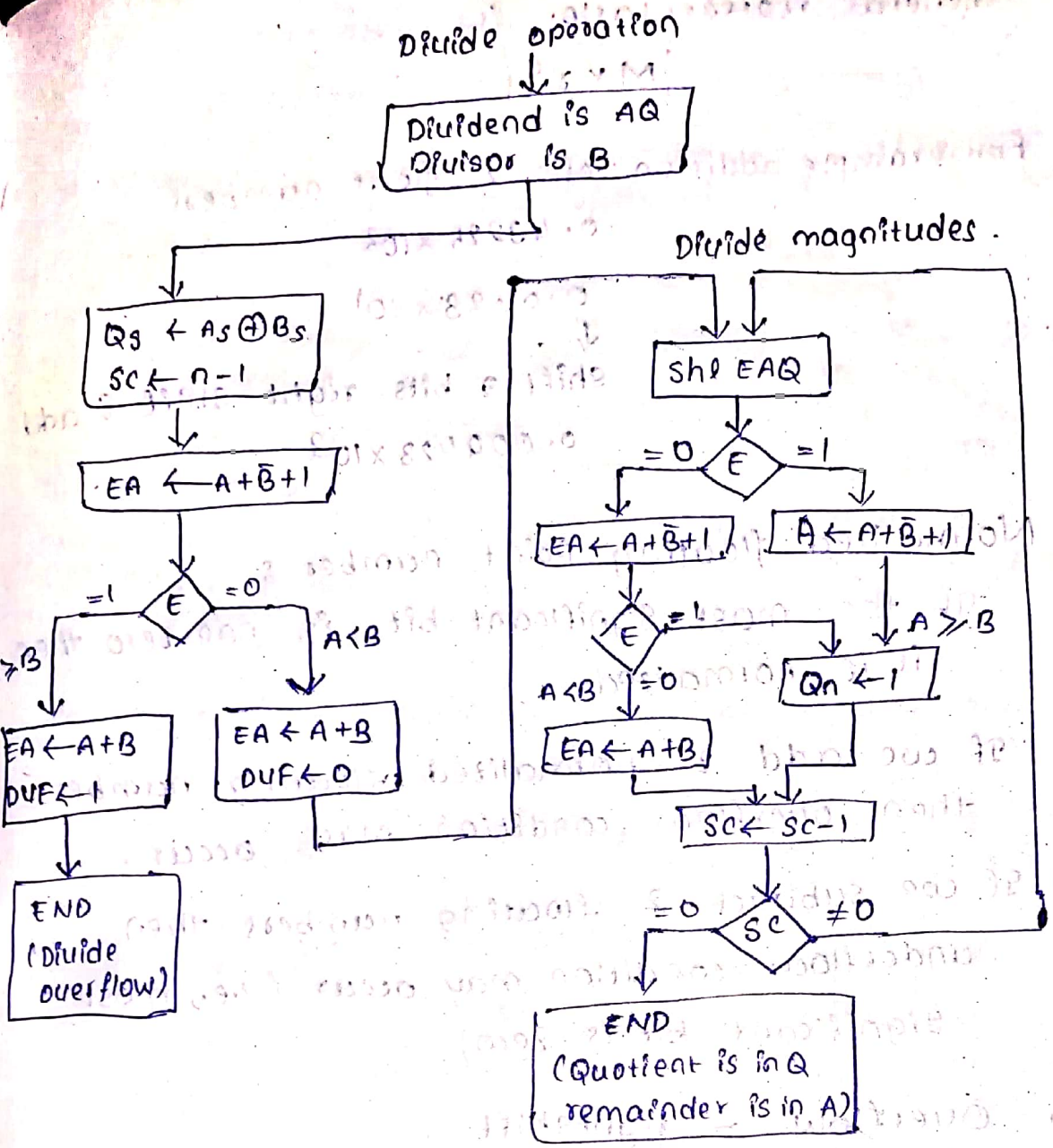
Quotient = 11010

Divide Overflow

⇒ Quotient overflow (when A > divisor)
that overflow bit is stored in flipflop

⇒ Drawback :- Time consuming as we have to transfer Q register value as well as flipflop information.

⇒ Divide overflow & divide by zero condition
* calls subroutines
* terminates the program
* raise interrupt request
* display warning messages.



Floating point representation

It contains 2 parts

⇒ Fraction part (Mantisa) → signed bit fixed point numb

⇒ Exponent part

↓

It specifies the position of the decimal point in the number.

+432.225

↓

It can be represented in system as

0.432225×10^3

↓

mantessa

General representation is

$M \times r^e$

For example addition of 2 float numbers

0.43225×10^3

0.00423×10^1

↓ shift 2 bits right shift & add

0.000423×10^3

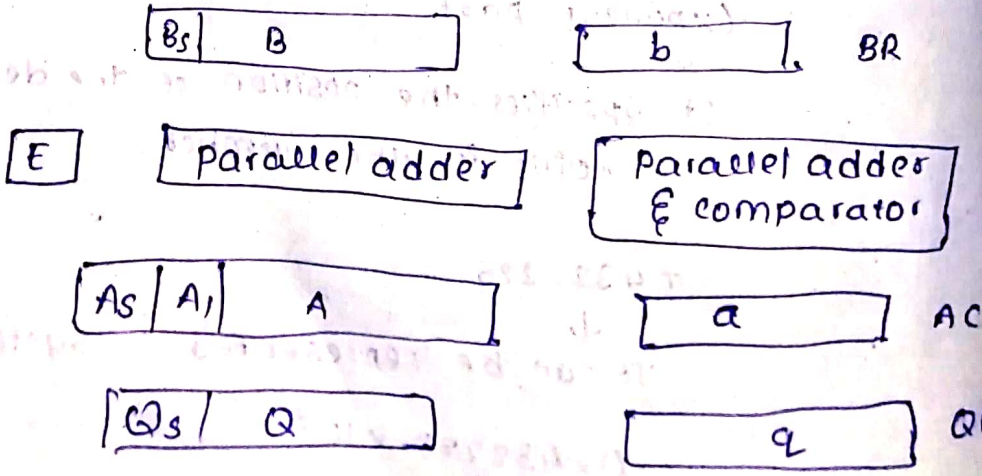
⇒ Normalized floating point number :-
If the most-significant bit is non-zero then it is normalized.

⇒ If we add 2 normalized floating numbers then overflow condition may occur.

⇒ If we subtract 2 floating numbers then underflow condition may occur (i.e., most significant bit is zero)

⇒ Overflow — rightshift
underflow — leftshift

Register configuration :-



Registers for floating point arithmetic operations

⇒ A_1 bit represents the most significant digit

⇒ B & A — mantissa

⇒ If $A_1 = 1$, result has to be normalized

⇒ a & b — exponent.

⇒ 1st parallel adder is used for adding mantissa values.

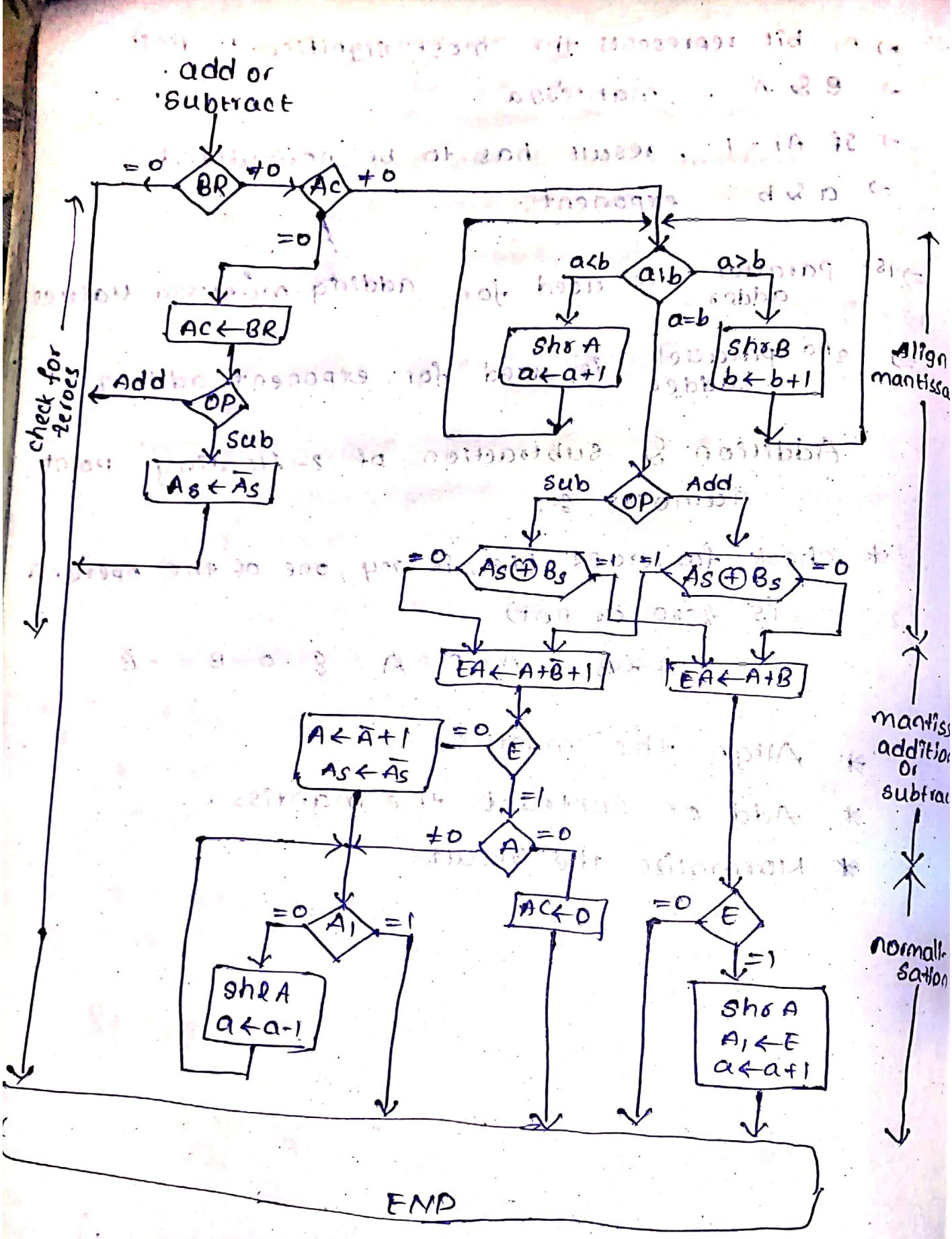
⇒ 2nd parallel adder is used for exponent adding

Addition & subtraction of 2-floating point numbers :-

* check for zeroes (i.e., if any one of the operand is zero or not)

~~A - B~~ $A - 0 = A$ & $0 - B = -B$

* Align the mantissa
* Add or subtract the mantissa
* Normalize the result.



Multiplication

The multiplication algorithm can be subdivided into 4 parts :-

1. check for errors
2. Add the exponents
3. Multiply the mantissas
4. Normalize the product