II yr - CSE
IV Sem.

UNIT-II. Arithmetic FOR COMPUTERS.

① Explain Sequential Multiplication & its Hardware Implementation (Apl may-18, Nov/Dec-15, 16).

(*) Multiplication is a slightly more complex operation than addition or subtraction, being implemented by shifting as well as addition.

Sequential version of the Multiplication Alg & Hlw.

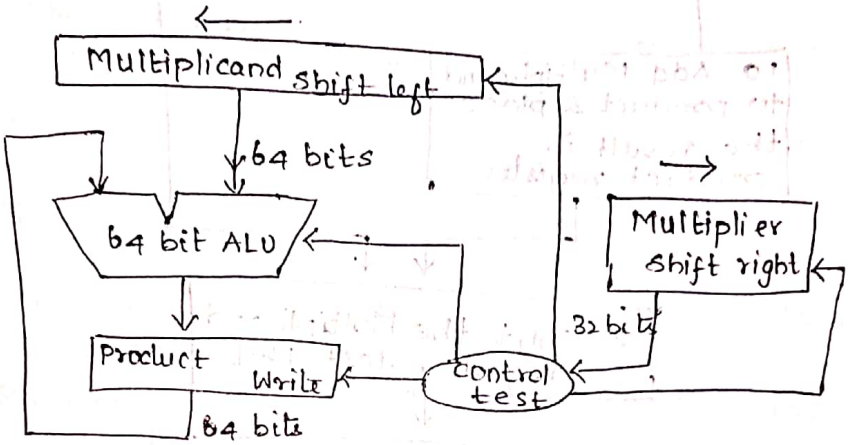


fig :- First version of Multiplication Hardware

→ It is constructed with 64 bit Mplicand, ALU and Product register and one 32 bit Mplier.

→ The 32 bit multiplier is loaded in 32 bit multiplier reg, 32 bit Mplicand is loaded in right half of 64 bit Multiplicand register and zero in the left half and the 64-bit product register is initialized to 0(zero).

→ The above said procedure is executed for 32 times to get 64-bit product result.

Algorithm:

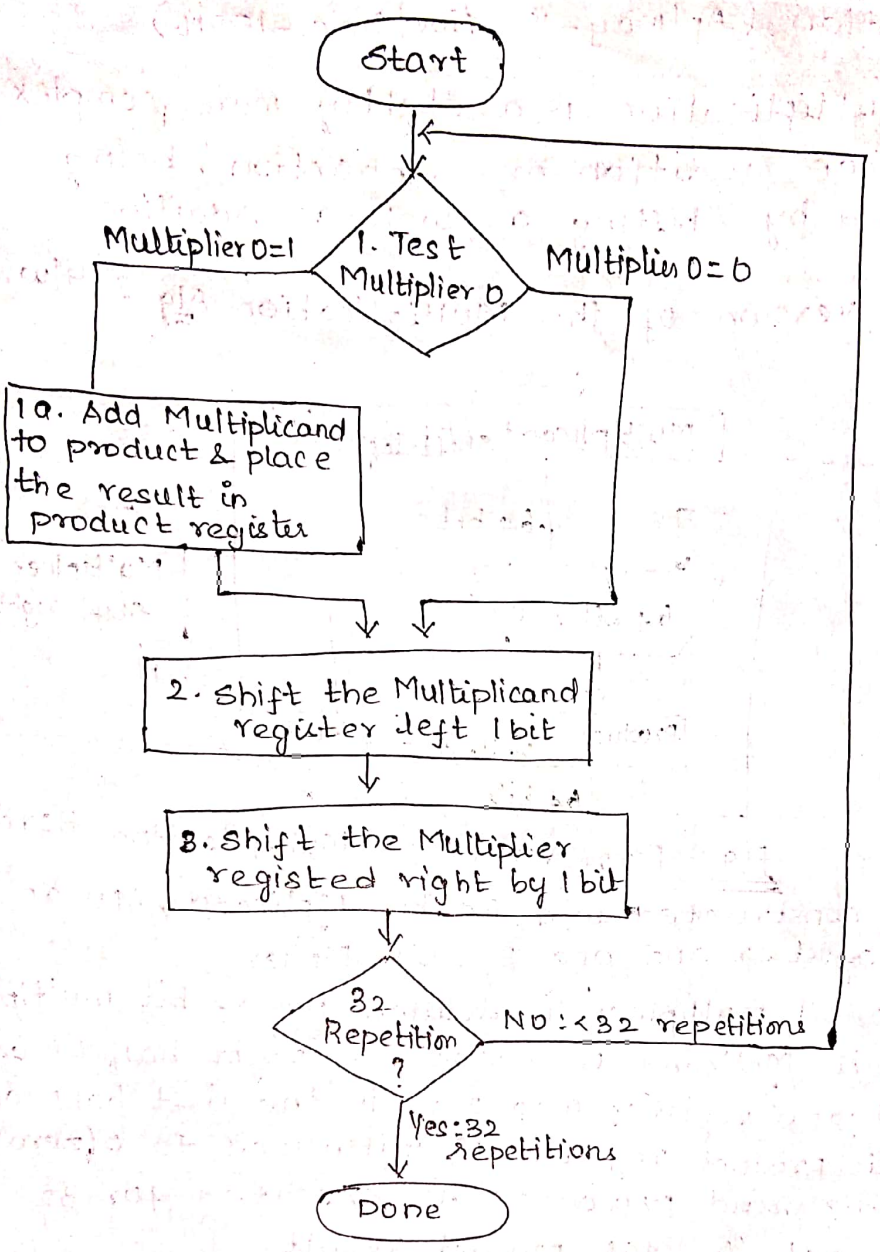
Step 1 : The LSB bit of the Multiplier 0 determines whether the Multiplicand is added to the product register.

Step 2 : shift the Multiplicand by 1 bit left logically that will have a effect of moving the intermediate operands to the left.

Step 3 : shift the Multiplier right logically by 1 bit that will give the next bit of the multiplier ~~bit~~ to be examined.

These 3 steps are repeated 32 times to get Product.

First/Sequential version of Multiplication Algorithm.



Example :-

Multiply $4_{10} \times 3_{10} = 12_{10}$.

(or) $0100_2 \times 0011_2 = 0000\ 1100_2$.

4 => Multiplicand 0100_2
3 => Multiplier, 0011_2

Iteration	Steps	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0100	0000 0000
1	1a : 1 → Product = Produ + Mcand	0011	0000 0100	0000 0100
	2. shift left Mcand	0011	0000 1000	0000 0100
	3. Shift right Multiplier	0001	0000 1000	0000 0100
2	1a : 1 → prod = prod + Mcand	0001	0000 1000	0000 1100
	2. Shift left Mcand	0001	0001 0000	0000 1100
	3. Shift right Multiplier	0000	0001 0000	0000 1100
3.	1a: 0 → No operation	0000	0001 0000	0000 1100
	2. Shift left Mcand	0000	0010 0000	0000 1100
	3. Shift right Multiplier	0000	0010 0000	0000 1100
4.	1a : 0 → No operation	0000	0010 0000	0000 1100
	2. Shift left Mcand	0000	0100 0000	0000 1100
	3. Shift right Multiplier	0000	0100 0000	0000 1100

From the above model it is observed that

(*) If each step took a clock cycle, this alg would require almost 100 clock cycle

(*) half bits in Multiplicand always 0, Thus, 64-bit adder is wasted.

(*) 0's inserted in left half Multiplicand as shifted.
↳ Least significant bits of Product never change once formed, instead of shifting Multiplicand to left, shift product to right (ie Refined version).

② Explain booth Multiplication with example. Apl May 15

BOOTH's Algorithm:-

=> It is a powerful alg for signed-number multiplication, which treats both +ve and -ve nos uniformly.

=> In this algorithm, multiplier bits are encoded from de right to left, before the bits are used for getting partial products.

=> To encode the multiplier bits, two adjacent bits bi (current bit) & bi-1 (previous bit) is examined as:

Multiplier		Version of Mcand
Bit i	Bit i-1	selected by bit i
0	0	0 x M
0	1	+1 x M
1	0	-1 x M
1	1	0 x M

Booth Multiplier Recoding table.

Worst-case Multiplier: 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

⇓

+1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1

Ordinary multiplier: 1 1 0 0 0 1 0 1 1 0 1 1 1 1 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 1 0 0 0 0 1 1 1

⇓

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

Ex:- Multiply 001100 (+12) & 010011 (+19).

Multiplier:- 0 0 1 1 0 0 (12) [0] Mcand:- 0 1 0 0 1 1 (19)

Recoded Multiplier:- 0 +1 0 -1 0 0

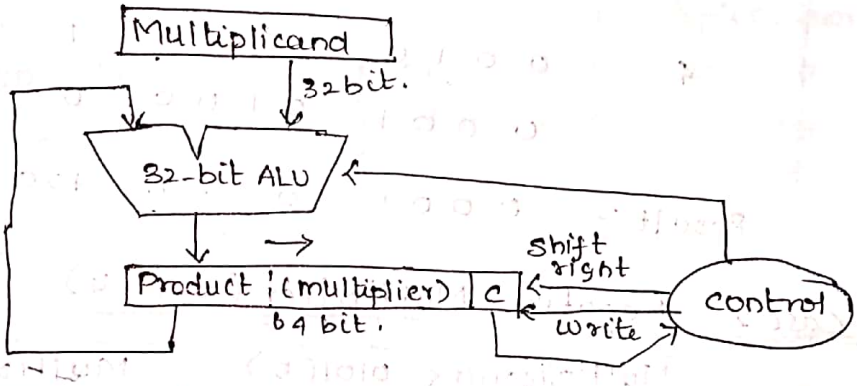
```
        0   1   0   0   1   1
      x
        0  +1   0  -1   0   0
-------------------------------
  0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0
  1 1 1 1 1 0 1 1 0 1        <- 2's complem of Mcand
  0 0 0 0 0 0 0 0 0
  0 0 0 1 0 0 1 1
  0 0 0 0 0 0 0
-------------------------------
  0 0 0 0 1 1 1 0 0 1 0 0   (228)
```

Hardware Implementation of Booths Algorithm.

Other ex:- ① +14 (01110) and -5 (11011)

② -13 (110011) and -20 (101100)

Hardware Implementation of Booths Algorithm:



→ The hardware consists of 32-bit register M for the multiplicand, 64-bit product register (PR), and a 1-bit register C, 32-bit ALU & control.

→ Initially, M contains Multiplicand, PR contains multiplier in lower half of register and

↳ & initialize the upper half of product register to 0, & C contains bit 0.

The alg is the following steps:

Repeated 32 times:

1. If (Q0, C) pair is:
   - → 10 : PR = PR − M
   - → 01 : PR = PR + M
   - → 00 & 11 No operation.
2. Arithmetic shift PR right by 1 bit. The shift-out bit, gets into C.

Case 1: Both positive (5 * 4)

Multiplicand ← 0101 (5)			Multiplier ← 0100 (4)	
Steps	PR		C	Operation
	0 0 0 0	0 1 0 0	0	Initial. 00 => No operation shift right
1.	0 0 0 0	0 0 1 0	0	00 => No operation Shift right PR.
2.	0 0 0 0	0 0 0 1	0	00 => No operation Shift right PR.
3	1 0 1 1	0 0 0 1	0	10 => PR = PR − M.
	1 1 0 1	1 0 0 0	1	Shift right.
4.	0 0 1 0	1 0 0 0	1	01 => PR = PR + M.
	0 0 0 1	0 1 0 0	0	Shift right
Result :-	0 0 0 1	0 1 0 0	+20.	

Case 2: Negative Multiplier (5 * −4)

Multiplicand ← 0101 (5)			Multiplier ← 1100 (−4)	
Steps	PR		C	Operation
	0 0 0 0	1 1 0 0	0	Initial
step 1	0 0 0 0	0 1 1 0	0	00 → No operation Shift right
step 2	0 0 0 0	0 0 1 1	0	00 → No oper Shift. right
step 3	1 0 1 1	0 0 1 1	0	10 → PR = PR − M
	0 1 1 0 1	1 0 0 1	1	Shift right
step 4	1 1 0 0	1 1 0 0	1	11 → No oper Shift right
Result	1 1 1 0	1 1 0 0	(−20)	(2's comp of 20)

Case 3: Negative Multiplicand (-5 * 4)

Multiplicand ← 0101 (5) Multiplier ← 0100 (4)

Steps	PR		C	Operations
	0 0 0 0	0 1 0 0	0	Initial
		=	=	
& 1.	0 0 0 0	0 0 1 0	0	00 → No operation
		=	=	shift right
2.	0 0 0 0	0 0 0 1	0	00 → No operation,
		=	=	shift right.
3.	0 1 0 1	0 0 0 1	0	10 → PR = PR - M.
	0 0 1 0	1 0 0 0	1	shift right
		=	=	
4	1 1 0 1	1 0 0 0	1	01 → PR = PR + M
	1 1 1 0	1 1 0 0	0	shift right

Result : 1 1 1 0 1 1 0 0 0 = -20 (2's comp of 20)

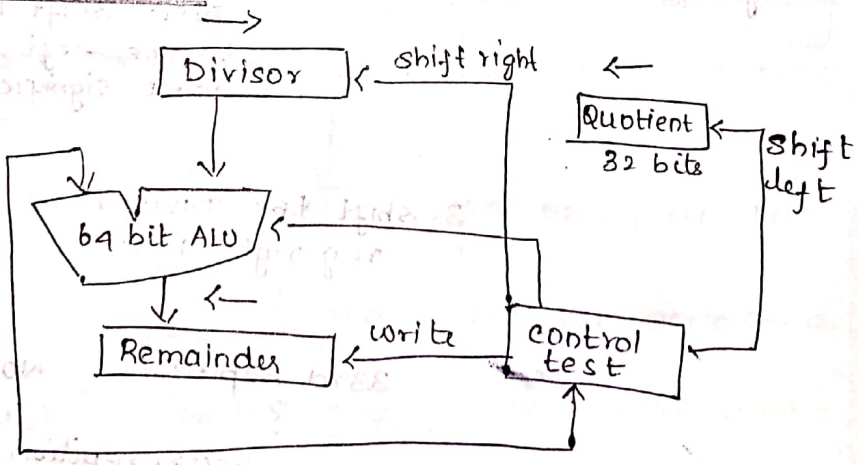
---

(3) Explain about Division alg with an example.

(*) Division is reciprocal operation of multiplication.

(*) Divide two operands dividend and divisor and produce the results, called the quotient and remainder.

A. division Alg & Hardware:



Division hardware.

(*) The Divisor register, ALU & Remainder reg are all 64 bits wide, with only the Quotient register being 32 bits.

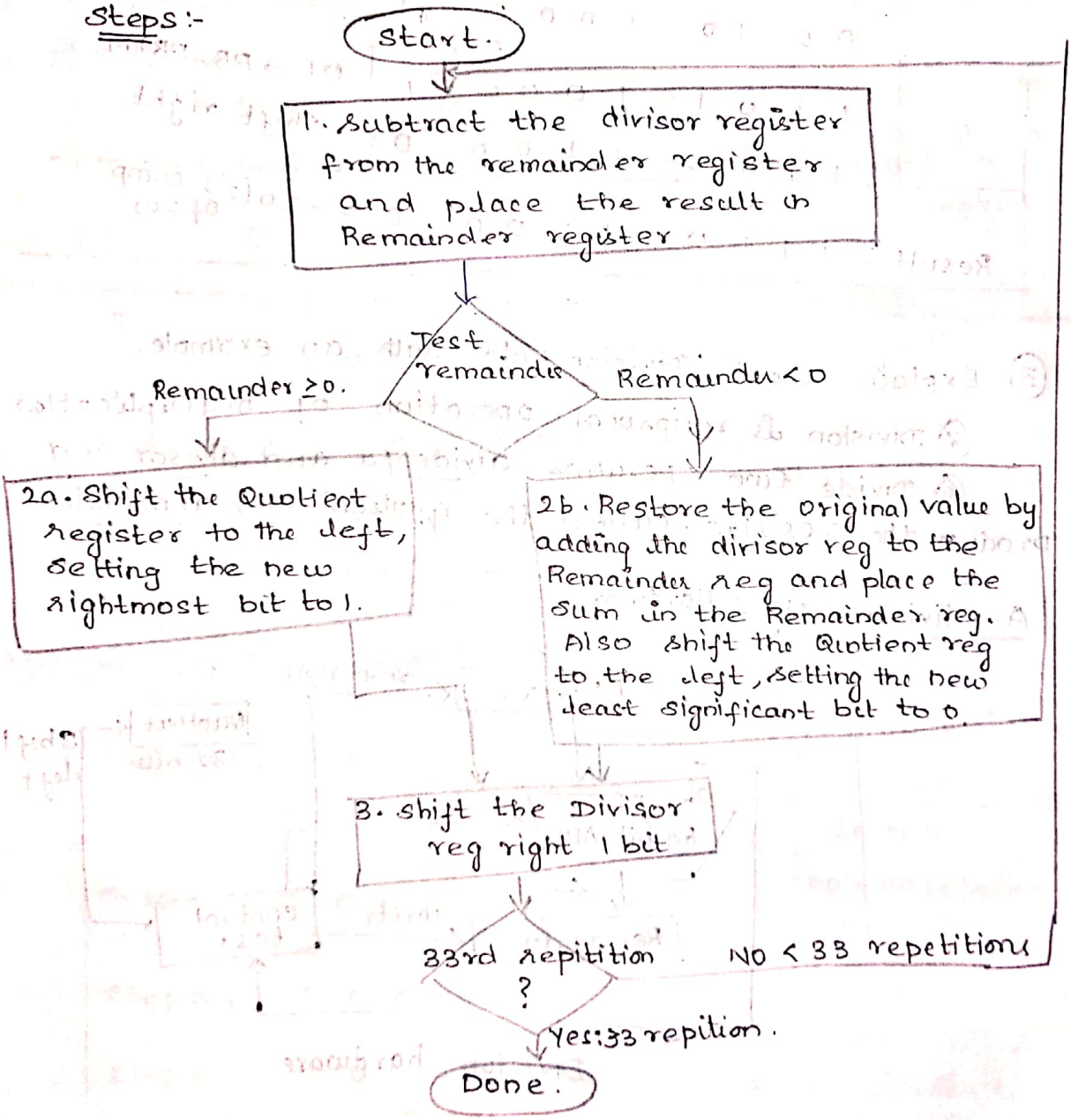
(*) The 32 bit divisor starts in the left half of the divisor reg & is shifted right 1 bit each iteration.

(*) The 32 bit quotient register is set to 0.

(*) The remainder is initialized with the divident.

(*) Control decides when to shift the divisor & quotient & when to write the new value into the remainder reg.

Steps :-



Divide : 8 x 3

8 -> dividend

3 -> divisor

Ex: Divide 8_{10} by 3_{10} $0000\ 1000_2$ by 0011_2 using divisin Alg.

Div → Divisor

Iteration	Steps	Quotient	Divisor	Remainder
0	Initial Values.	0000	0011 0000	0000 1000
1	1. Rem = Rem − Div	0000	0011 0000	1101 1000
	2b. Rem<0 → Rem+Div, sll $Q_1 Q_0 = 0$	000 0	0011 0000	0000 1000
	shift right Divisor Reg	000 0	0001 1000	0000 1000
2.	1. Rem = Rem − Div	0000	0001 1000	1111 0000
	2b. Rem<0 → Rem = Rem+Div sll $Q_1 Q_0 = 0$.	0000	0001 1000	0000 1000
	shift Right Divisor reg	0000	0000 1100	0000 1000
3.	1. Rem = Rem − Div	0000	0000 1100	1111 1100
	2b. Rem<0 → +Div, sll $Q_1 Q_0 = 0$	0000	00001100	0000 1000
	shift right divisor reg	0000	0000 0110	0000 1000
4.	1. Rem = Rem − Div	0000	0000 0110	0000 0010
	2a. Rem >0 → sll $Q_1 Q_0 = 1$	0001	0000 0110	0000 0010
	shift right Divisor reg	0001	0000 0011	0000 0010
5.	1. Rem = Rem − Div	0001	0000 0011	1111 1111
	2a. Rem<0 → +Div; sll $Q_1 Q_0 = 0$	0010	0000 0011	0000 0010
	shift right divisor Reg	0010	0000 0001	0000 0010
		Quotient (2).		Remainder (2)

(5) Explain about Restoring division with eg.

Alg:- (or) Second version of dirision.

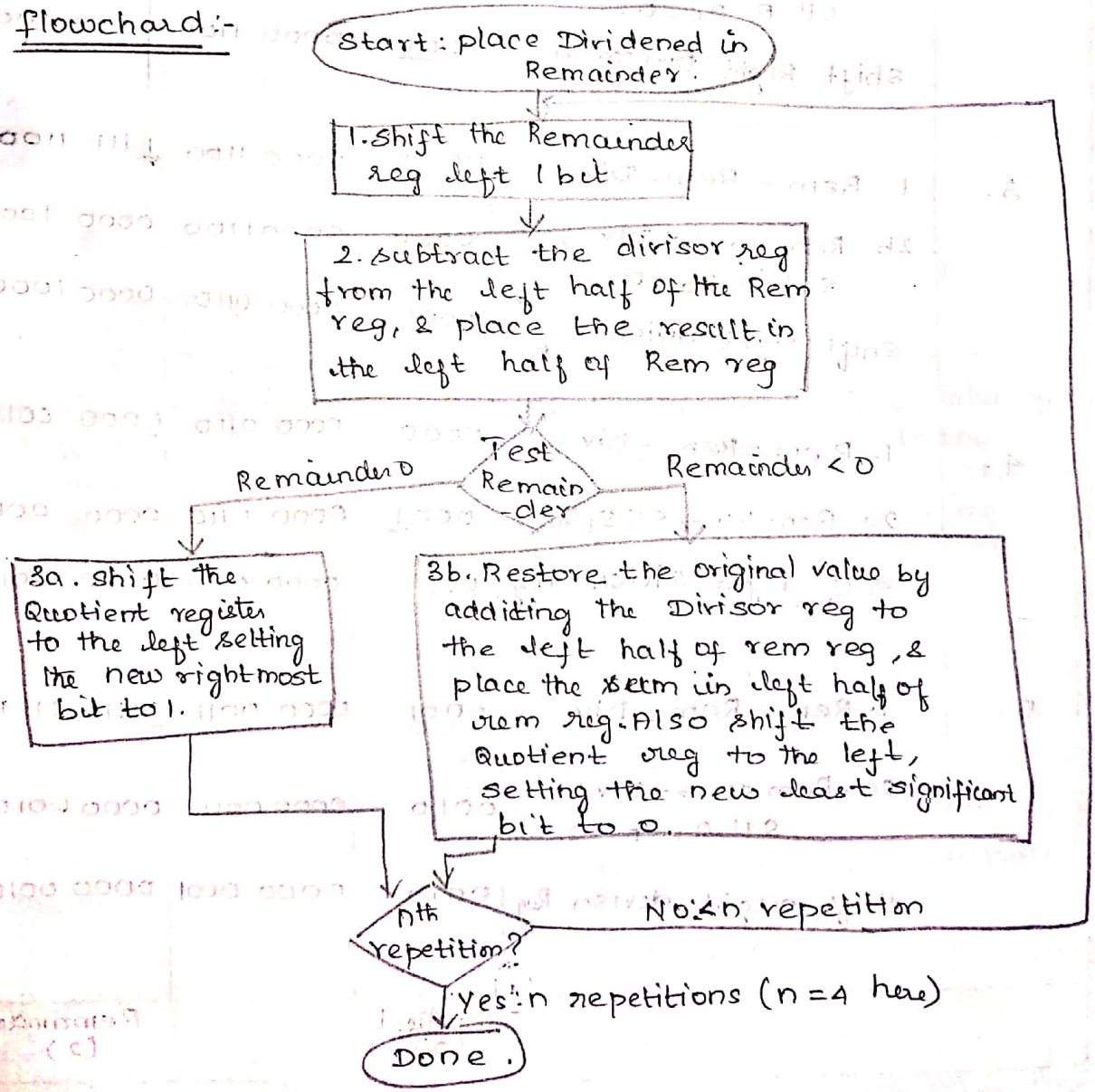
Step 1: Shift Remainder reg left logically (sll) one binary position

Step 2: Subtract divisor from left (upper half) of the remainder reg and the result in the left half of the remainder reg

Step 3: If the sign bit of Remainder reg is 1, then shift Quotient reg (Q) left by on binary position and set Q_0 to 0 and add divisor back to remainder reg (i.e., restore remainder reg); otherwise, set Q_0 to 1.

Step 4: Repeat steps 1, 2 and 3 for <33 times to obtain the remainder and quotient result.

flowchard:-



Ex:- 10_{10} by 3_{10} or $0000\ 1010_2$ by 0011_2. using Restoring Division. ↓ dividend. 0011 → divisor

Iteration	Steps	Q (quotient)	Remainder (Rem)	
			Left of the rem reg set to 0	Dividend
0	Initial values	0000	0000	1010
1.	Sll Rem Reg.	0000	0001	0100
	Left half Rem reg ← left Rem − Divisor		1101	
			1110	
			0011 +	
	Rem < 0, restore Rem Reg (+divisor), Sll Q, Q_0=0	0000	0001	0100.
2.	Sll Rem Reg	0000	0010	1000
	Left half Rem Reg ← left Rem − Divisor		1101	
			1111	
			0011 (+)	
	Rem < 0, restore Rem. Reg (+divisor), Sll Q, Q_0=0	0000	0010	1000.
3.	Sll Rem Reg.	0000	0101	0000
	Left half Rem Reg ← left Rem − Divisor		1101	
			0010	
	Rem > 0, Sll Q, Q_0=1	0001	0010	0000
4.	~~Rem~~ Sll Rem Reg	0001	0100	0000
	Left half Rem ← left Rem − Divisor.		1101	
			0001	
	Rem > 0, Sll Q, Q_0=1	0011	0001	0000
		Quotient (3)	Remainder (1)	

(5) (5) Explain Floating point representation and Subword parallelism in detail.

Floating point Representation:

Floating point:- Programming languages support numbers with fractions, which are called reals in mathematics.

eg:- $3.141\ldots_{10}$ (PI)

$2.71828\ldots_{10}$ (e)

0.000000001_{10} (or) $1.0_{10} \times 10^{-9}$ (sec in a nanosec)

$3,155,760,000_{10}$ (or) $3.15576_{10} \times 10^{9}$ (sec in a typical mem)

Scientific Notation:-

It is a notation that single digit to the left of decimal point.

Normalized Number:

A No in scientific notation that has no leading 0s is called normalized No.

eg:- $1.0_{10} \times 10^{-9}$ is a normalize scientific notation

$0.1_{10} \times 10^{-9}$ & 10.0×10^{-10} are not.

also

We can show binary Nos in scientific notation:

$1.0_2 \times 2^{-1}$.

→ Nos in which the binary pt is not fixed is called floating pt.

→ A std scientific notation for reals is normalized form has 2 adv. (1) It simp

Adv of Scientific notation:

(1) It simplifies exchange of data that includes floating point nos

(2) It increases the accuracy of the nos that can be stored in word.

Floating Point Representation:-

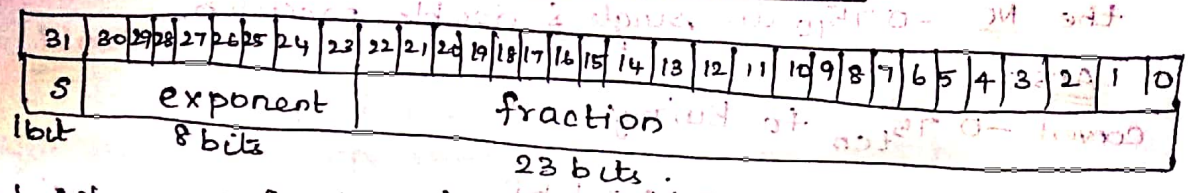
Floating-pt nos are usually a multiple of the size of a word.

The two floating point representation are

(i) Single precision Floating pt No.

(ii) Double " " " "

(a) Single - precision Floating point No :-



→ where S is the sign of the floating pt no (1 meaning -ve)

→ Exponent is the value of 8-bit exponent field (including the sign of the exponent), and

→ Fraction is the 23-bit No.

In general, floating-pt Nos are of the form:

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

The above representation is called single precision floating pt format.

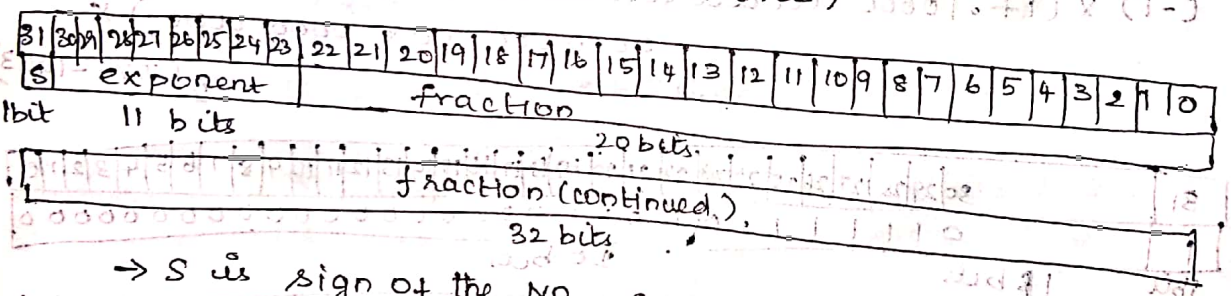
⇒ Overflow here means that the exponent is too large to be represented in the exponent field.

⇒ Underflow occur when -ve exponent is too large to fit in the exponent field.

(b) Double precision floating point No.

→ The operations on doubles are called double precision floating-point arithmetic.

→ It takes two MIPS words,



→ S is sign of the No, exponent is a value of 11-bit and fraction is the 52-bit No in the fraction field.

Advantage of double precision:

(1) Increase the exponent range

(2) Increase the precision because of the much larger fraction.

Format :-

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

Eg:- Show Convert IEEE 754 binary representation of the No -0.75_{10} in single & double precision...

Ans:-

Convert -0.75_{ten} to binary.

$0.75 \times 2 = 1.50$

$0.50 \times 2 = 1.00$

$\Rightarrow 0.11 \times 2^{-0}$

After normalizing : 1.1×2^{-1}

The general representation for a single precision No is

$$(-1)^S * (1 + \text{Fraction}) \times 2^{(\text{Exp}-127)}$$

Subtracting the bias 127 from the exp 1.1×2^{-1} yields:

$$(-1)^1 \times (1 + .10000000000000000000000_{two}) \times 2^{(126-127)}$$

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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit | 8 bits | 23 bits

The double precision representation is:

$(-1)^1 \times (1 + .10000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 0000_2) \times 2^{(1022-1023)}$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit | 11 bits | 20 bits

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

32 bits

(6) Explain steps in floating point addition and Multiplication:

Floating point addition:

Most computer uses dedicate hlw to run floating point operation as fast as possible

Steps:-

(1) Align the decimal No of the smallest exponent No that matches the larger exponent. (shift the smaller No right until exponent match)

(2) Add/subtract the mantissas depending on sign

(3) Normalise the sum by adjusting exponent

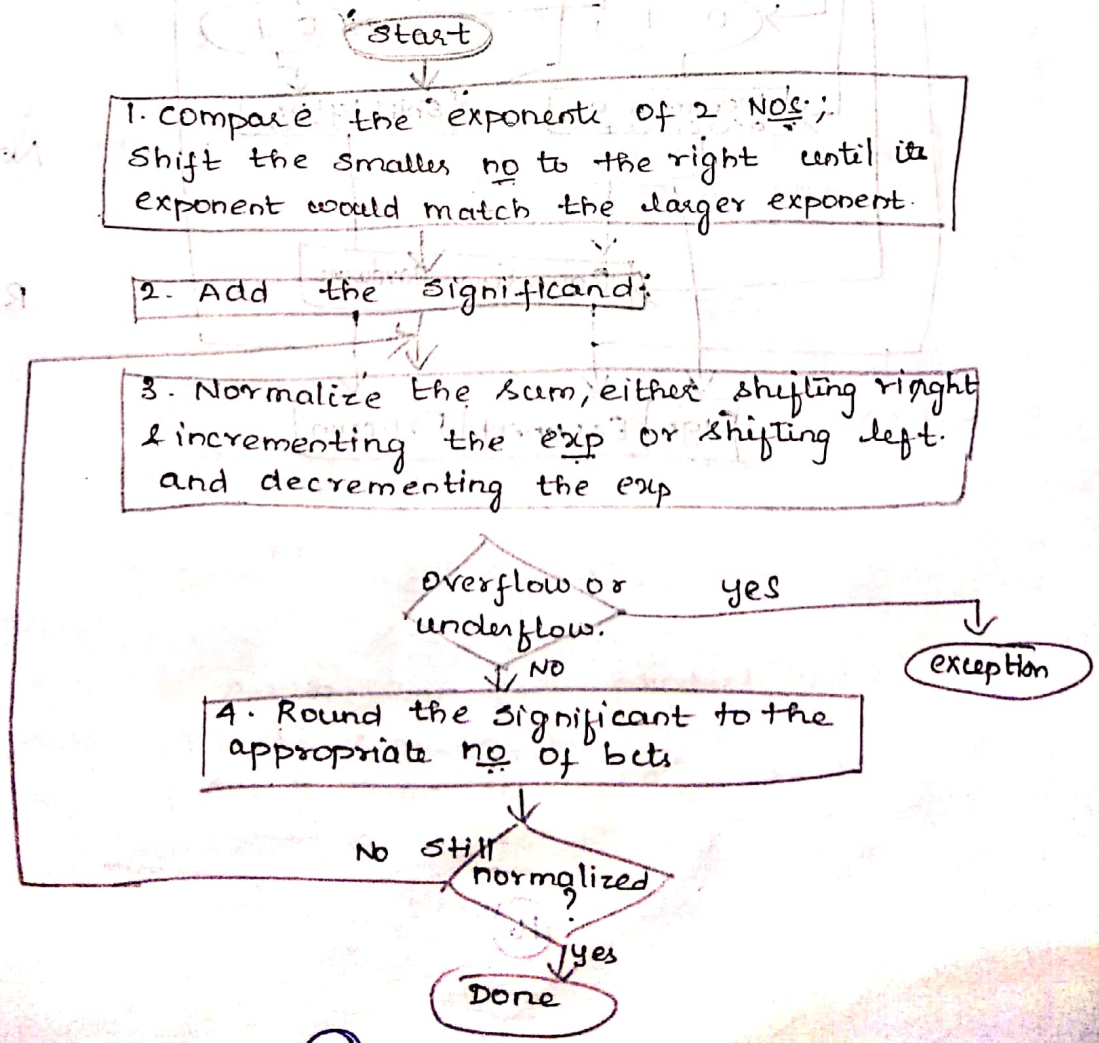
(4) Check for overflow.

(5) Rounding off to appropriate Nos of bit.

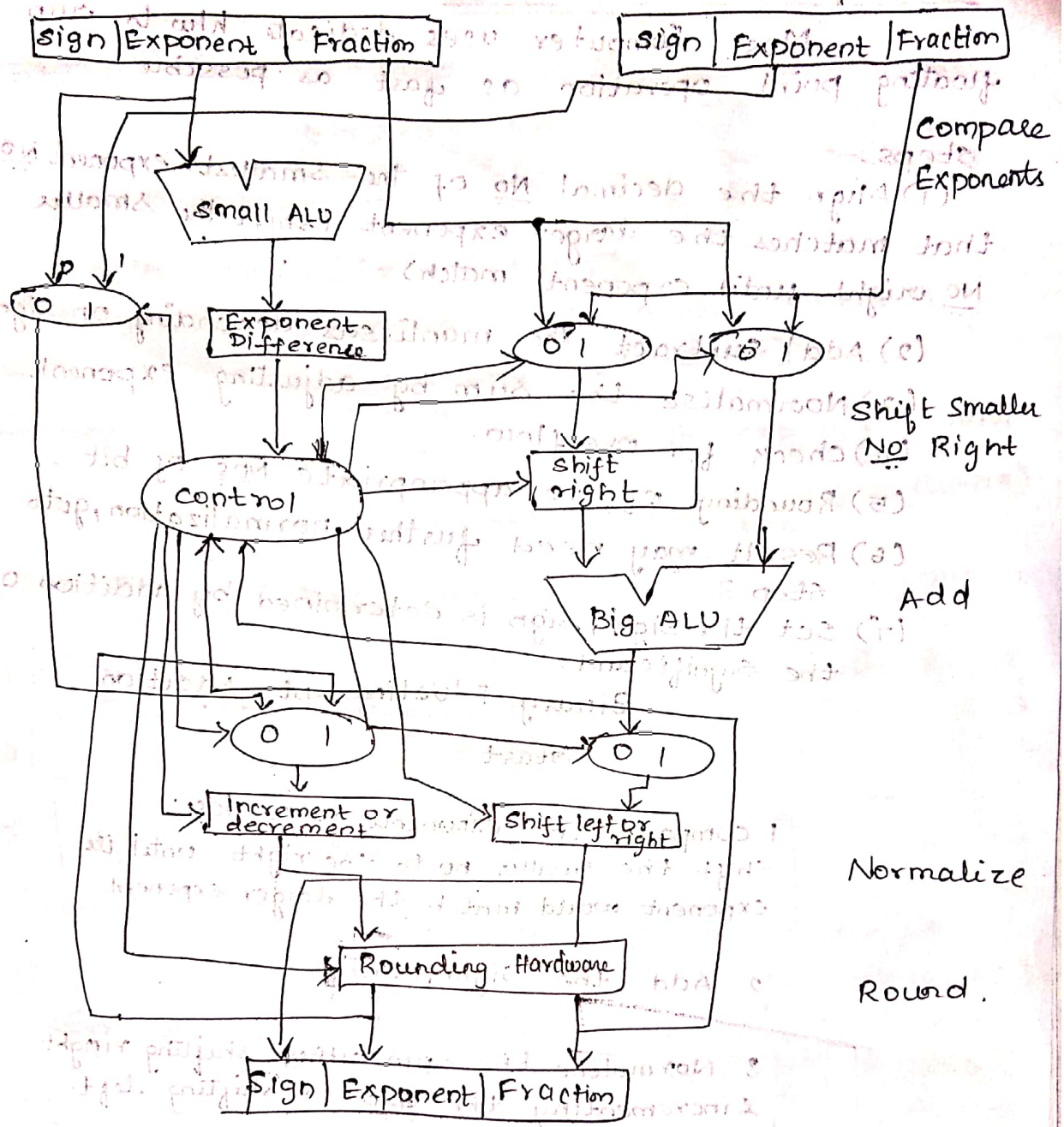
(6) Result may need further normalization, goto step 3

(7) Set the sign; sign is determined by addition of the significand.

Binary Floating -pt. Addition.



Block diagram of Floating point Addition.



Floating point Multiplication

Step1: Unlike addition, we can calculate the exp of product by simply adding the exp of the operands together.

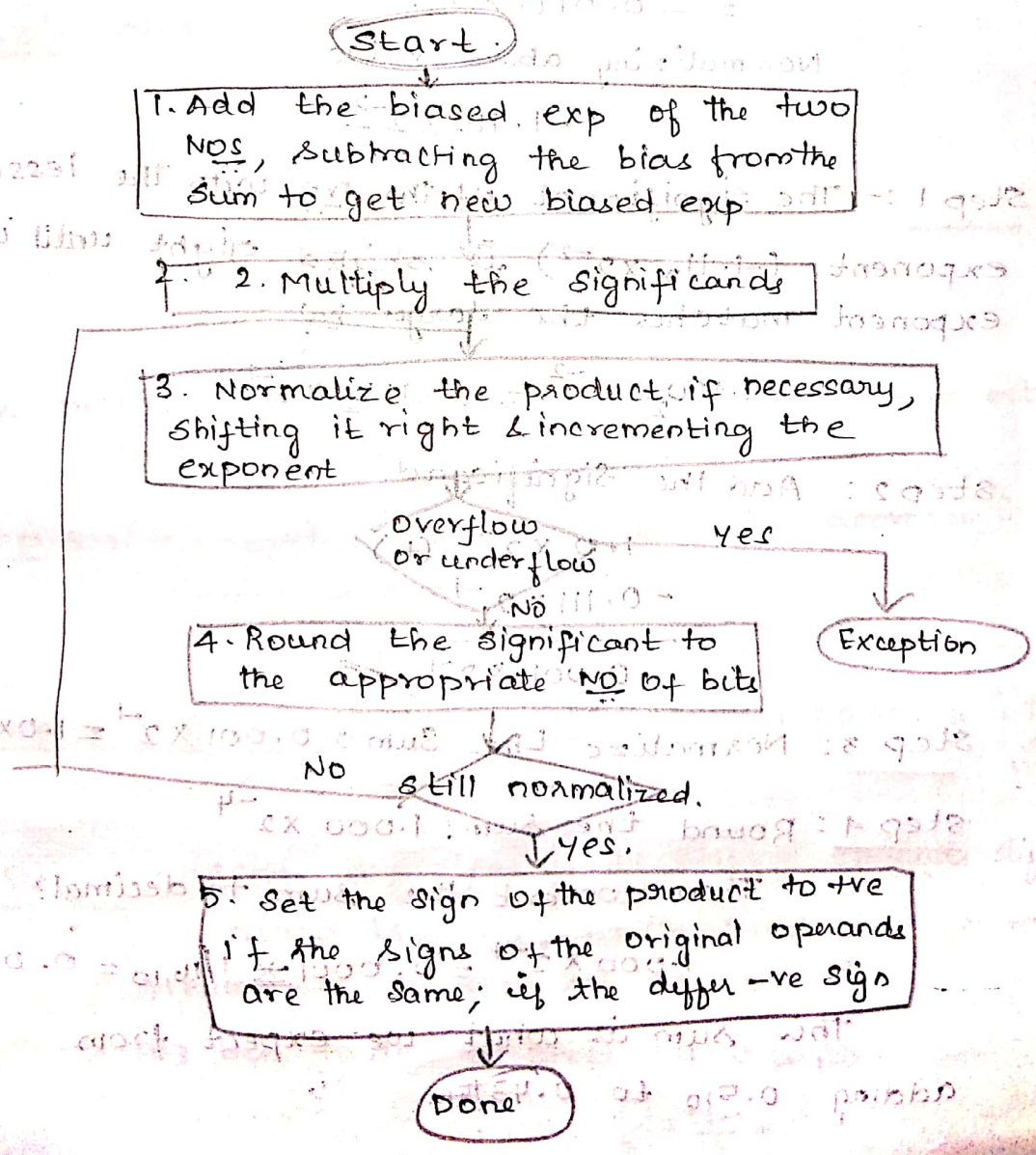
To get correct biased sum when we add biased Nos, we must subtract the bias from the sum.

Step 2: Next comes the multiplication of the significands.

step 3: Then if product is not normalized, normalize it

Step 4: Round of the product.

Step 5: The sign of product depends on the ~~original~~ signs of original operand.



④ Add & Multiply 0.5_{10} & -0.4375_{10} using floating point.

Floating point addition:-

convert dec to binary

0.5_{10} $0.5 \times 2 = 1.0$

Normalizing above value : $= 1.0 \times 2^{-1}$

-0.4375_{10} => $0.4375 \times 2 = 0.8750$

$0.8750 \times 2 = 1.7500$

$0.7500 \times 2 = 1.5000$

$0.5000 \times 2 = 1.0000$

$= -0.0111 \times 2^{0}$

Normalizing above value

-1.110×2^{-2}

Step 1 :- The significant of the no with the lesser exponent ($-1.11_2 \times 2^{-2}$) is shifted right until its exponent matches the larger no

$-1.110_2 \times 2^{-2} = 0.111_2 \times 2^{-1}$

Step 2: Add the significand

$$\begin{array}{r} 1.0 \times 2^{-1} \quad (+) \\ -0.111 \times 2^{-1} \\ \hline 0.001 \times 2^{-1} \end{array}$$

Step 3: Normalize the sum : $0.001 \times 2^{-1} = 1.0 \times 2^{-4}$

Step 4: Round the sum: 1.000×2^{-4}

Then convert the sum to decimal:

$1.000 \times 2^{-4} = 0.0001_2 = 1/16_{10} = 0.0625_{10}$.

This sum is what we expect from adding 0.5_{10} to 0.4375_{10}.

Floating pt Multiplication.

$0.5_{10} \times (-0.4375)_{10}$

$0.5_{10} = 1.000 \times 2^{-1}$

$-0.4375_{10} = -1.110 \times 2^{-2}$.

Step 1: Adding the exponents without bias:

$$-1 + (-2) = -3$$

(or) using biased representation

$$(-1+127) + (-2+127) - 127$$
$$= (-1-2) + (127+127-127)$$
$$= -3 + 127 = 124.$$

Step 2: Multiply the significand:

$$\begin{array}{r} 1.000_{two} \\ \times\ 1.110_{two} \\ \hline 0000 \\ 1000 \\ 1000 \\ 1000 \\ \hline 1.110000 \end{array}$$

The product is 1.110000×2^{-3}, but we use 4 bits, so 1.110×2^{-3}

Step 3: Normalize the product. The above value is in normalized form.

$$1.110 \times 2^{-3}.$$

Step 4: Rounding the product makes no change in this eg value

$$1.110 \times 2^{-3}$$

Step 5: Since the signs of the original operands differ, make the sign of the product negative.

Hence the product is -1.110×2^{-3}.

Converting bin to dec to check our results:

$$-1.110 \times 2^{-3} = -0.00111 = \frac{1}{8} + \frac{1}{16} + \frac{1}{32}_{ten}$$
$$= -0.21875_{ten}$$