COMPUTER SCIENCE

Computer Organization and Architecture

ALU & Control unit

Lecture_01

Vijay Agarwal sir

01 IEEE 754 Floating Point Representation

02 Micro Operation

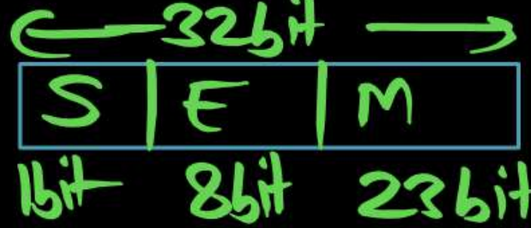
Floating Point Representation.

$E = e + bias$

m: Mantissa

IEEE 754 Floating point Representation.

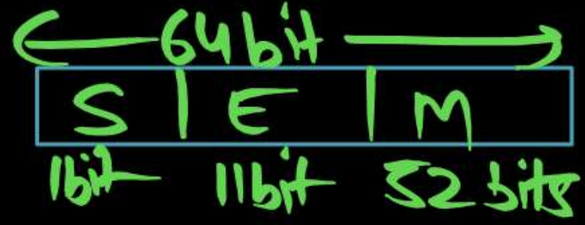
Single Precision



$$bias = 2^{8-1} - 1$$

Excess-127 | bias = 127

Double Precision.



$$bias = 2^{11-1} - 1$$

bias = 1023 Excess-1023.

MCQ

Consider the IEEE-754 single precision floating point numbers
P = 0xC1800000 and Q = 0x3F5C2EF4.
Which one of the following corresponds to the product of these numbers (i.e., P × Q), represented in the IEEE-754 single precision format?

[GATE-2023-CS: 2M]

A 0x404C2EF4

B 0x405C2EF4

C 0xC15C2EF4

D 0xC14C2EF4

Ans (C).

P = 0x C1800000

Q = 0x 3F5C2EF4

1 bit	8 bit	23 bit
S	E	M

$bias = 2^{8-1} - 1 \Rightarrow \boxed{bias = 127}$

P = 0x C1800000

1 100 0001 1000 0000 0000 0000 0000 0000

(E = 131)

Sign bit | E (8 bit) | Mantissa (23 bit)

S = 1 (-ve)

E = 10000011 = 131

$BE = AE + bias$

$E = e + bias$

$e = E - bias$

E = 131

bias = 127

$(-1)^S \; 1.M \times 2^e$

$(-1)^1 \; 1.0000000 \times 2^{131-127}$

BE = E = 131

$e = 131 - 127$

$e = +4$

M = 00000000

$\boxed{P = -(1.0000000) \times 2^{+4}}$

Q = 3F5C 2EF4

E=126 bias=127

e = E - bias

Sign = 0 (+ve)

E = 01111110 ⇒ E=126

BE ⇒ E = 126

$(-1)^S \; 1.M \times 2^e$

$(-1)^0 \; 1.101\,1100\,0010\,1110\,1111\,0100 \times 2^{126-127}$

M = 101 1100 0010 1110 1111 0100

bias = 127

E = e + bias

e = E - bias

126 - 127 = -1

$Q = (1.101\,1100\,0010\,1110\,1111\,0100) \times 2^{-1}$

$P = -1.0 \times 2^{+4}$

$Q = 1.101 \; 1100 \; 0010 \; 1110 \; 1111 \; 0100 \times 2^{-1}$

$P \times Q =$ Step 1 ADD the exponent $= +4 - 1 =$ $e = +3$

Step 2 : Multiply the Significant (Mantissa) $= 1 \times$

Step 3 Normalize (if Required)

Sign : -ve.

P x Q = exponent = (+4) + (-1) = +3.

Mantissa = (1.0000) * (1.101 1100 0010 1110 1111 0100)

$= -(1.101\ 1100\ 0010\ 1110\ 1111\ 0100) \times 2^{+3}$

M

Sign = 1 (-ve)

e = +3

bias = 127

E = e + bias

= 3 + 127

E = 130

E = 10000010

E = e + bias

= 3 + 127

E = 130

10000010

Sign (1 bit) | E (8 bit) | Mantissa (23 bit)

1 | 1000 0010 | 101 1100 0010 1110 1111 0100

12 12 14 15

C 1 5 C 2 E F 4

(C15C2EF4 H) Ans

if you don't know floating Point Multiplication or addition then also U can solve by Alternate Approach.

Alternate Approach

2nd Approach

Alternate Approach

P = C1B00000

1 bit	8 bit	23 bit
S	E	M

S(1bit) E(8bit) Mantissa (23bit)

1 10000011 10000000000000000000000

S = 1 (-ve)

E = 10000011

BE ⊗ E = 131

M = 00000000

$bias = 2^{8-1} - 1$

bias = 127

E = e + bias

e = E - bias

$(-1)^S \; 1.M \times 2^e$

$(-1)^1 \; 1.000000 \times 2^{131-127}$

$-1.000000 \times 2^{+4}$

-10000.00

P = -16

$-1.0000000000 \times 2^{+4}$

10000.0000

P = -16

$Q = 3F5C2EF4$

Sign	E (8 bit)	Mantissa (23 bit)
0	0111 1111	0101 1100 0010 1110 1111 0100

$e = E - bias$

$S = 0$ (+ve)

$E = 01111110$

$E = 126$

$m = 10111000010\ldots$

$bias = 127$

$E = e + bias$

$e = E - bias$

$(-1)^S \cdot 1.M \times 2^e$

$126 - 127$

$(-1)^0 \cdot 1.101\ 1100\ 0010\ 1110 \times 2$

$Q = 1.1011\ 1100 \times 2^{-1}$

$= 0.1101\ 1100$

$Q = 0.8593$

$Q = 0.8593$

p = -16 , q = 0.8593

p * q = -16 * 0.8593 = -13.75

(8) -(13.75)

-1101.11

$-1.10111 \times 2^{+3}$

e = +3

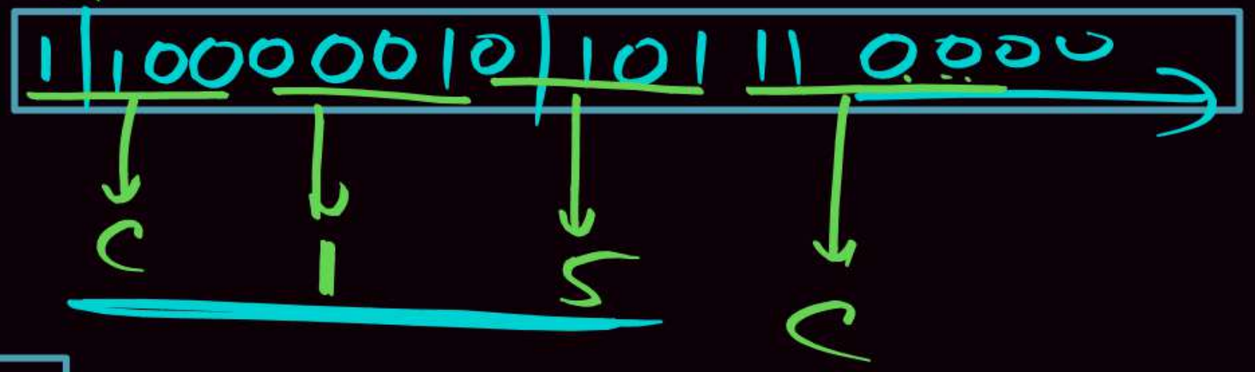
E = e + 127

E = 130

10000010

m: 101 11

Sign = 1



$P * Q = -16 \times .8593$

$= -(13.75)$

$P * Q = -13.75$

-1101.11

$\Rightarrow -1.101111 \times 2^{+3}$

$e = +3$ $\quad$ bias $= 127$

$E = e + bias \Rightarrow 3 + 127$

$= 130$

$E = 130$

$\Rightarrow 10000010$

$M = 101111$

$S = 1$ -ve

-16×0.8593

S(1bit) E(8bit) M(23bit)

1	10000010	10111.....

12 12

C . 1 5 C

$\Downarrow$

$(C15C2EF4)_h$ Ans

Why bias $2^{k-1} - 1$?

↳ Already explained in Previous Lecture.

IEEE 754 Floating Point Representation

IEEE 754 Floating Point Representation

Single Precision
(32 bit)
Excess 127

$bias = 2^{K-1} - 1$

$= 2^{8} - 1$

Bias = 127

Double Precision
(64 bit)
Excess 1023

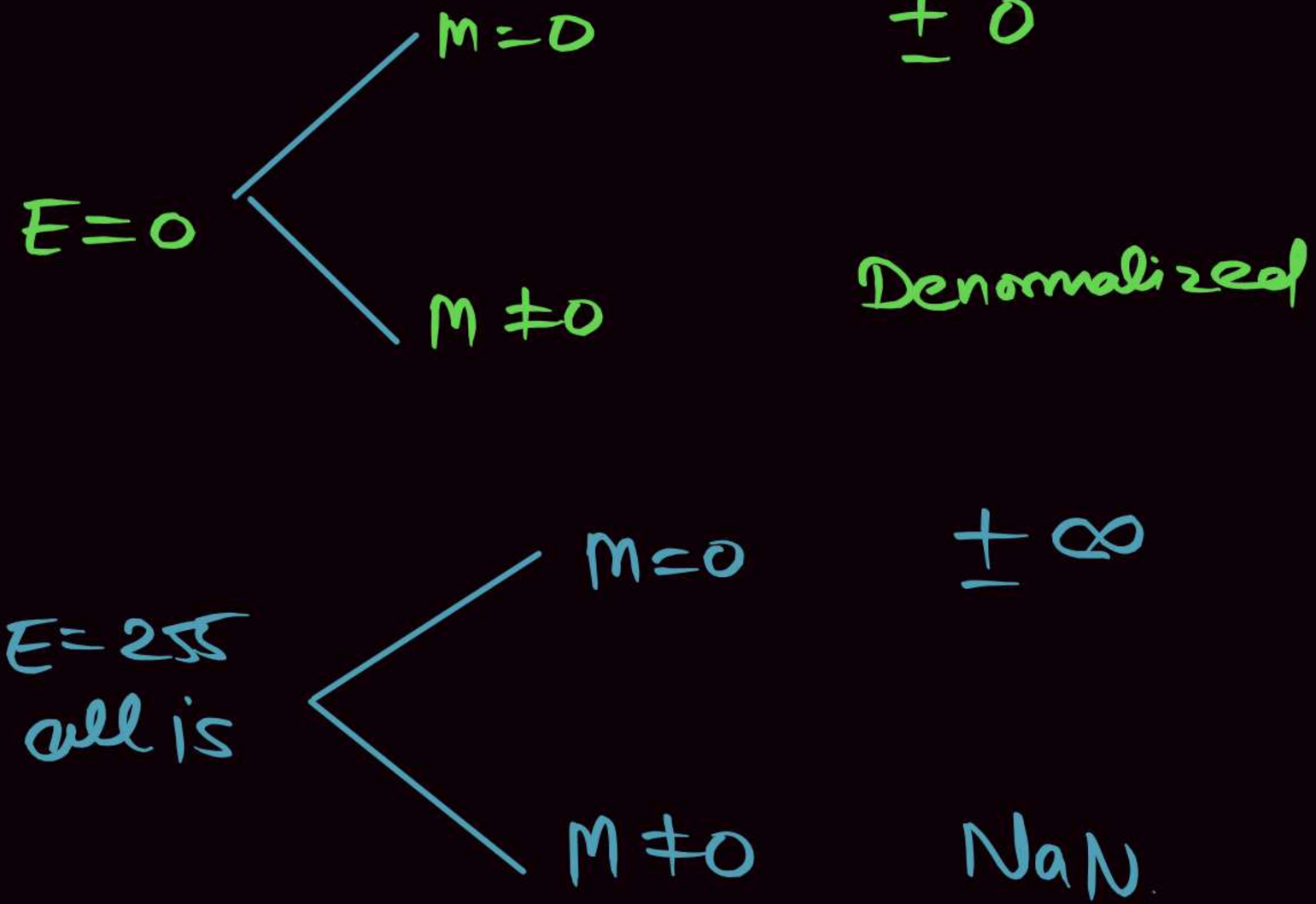
$bias = 2^{11-1} - 1$

Bias = 1023

Single Precision (32 bit)

S	E	M
1 bit	8 bit	23 bit

Sign(1 bit)	E(1 bit)	M(23 bit)	Value
0 or 1	00000000 E = 0	00000000000000000000000 M = 0	±0
0 or 1	11111111 E = 255	00000000000000000000000 M = 0	±∞
0 or 1	1 ≤ E ≤ 254	M = ...	Implicit Normalized form $(-1)^S \times 1.M \times 2^e$ $(-1)^S \times 1.M \times 2^{E-127 \text{ bias}}$
0 or 1	E = 0	M ≠ 0	Denormalized number/Fractional form $(-1)^S \times 0.M \times 2^{E-127 \text{ bias}}$
0 or 1	E = 255	M ≠ 0	Not a Number (NAN)


M=0
±0
E=0
M≠0
Denormalized
M=0
±∞
E=255
all is
M≠0
NaN

Double Precision

Excess - 1023

Sign (1 bit)	E(11 bit)	M(52 bit)	Value
0 or 1	0000 0000 000 E = 0	00000000000.. M = 0	± 0
0 or 1	1111 1111 111 E = 2047	0000000000.. M = 0	± ∞
0 or 1	1 ≤ E ≤ 2046	M = _ _ _ _ _ _ _ _ _ _	Implicit Normalization $(-1)^S \cdot M \times 2^e$ $(-1)^S \times 1 \cdot M \times 2^{E-1023}$
0 or 1	E = o	M ≠ 0	Denormalized number/ Fractional Form $(-1)^S 0 \cdot M \times 2^{E-1023}$
	E = 2047	M ≠ 0	Not a number

Denormalized Number.

NOTE: When $E = 0$ then Value 0 or fractional form

[when $M = 0$] [when $M \neq 0$]

Double Precision.

NOTE: When $E = 2047$
then Value ∞ or Not a Number(NAN)

[when $M = 0$] [when $M \neq 0$]

Note

Features IEEE 754 are special symbols to represent unusual events....

For example instead of interrupting on a divide by 0,softwarecan set the result to a bit pattern representing $+\infty$,- ∞ ; The largest exponent is reserved for these special symbols.

IEEE 754 has a symbols for the result of invalid operations, such 0/0, or subtract infinity from infinity . This symbols is NaN, for Not a Number.

The purpose of NaN is to allow programmer to postpone some test and decisions to a later time in this program. (when it is convenient)

Denormalized Number ?

$E = e + bias$

$E = e + 127$

Single Precision.

$bias = 127$

Denormalized Number

E=e+bias
e=E-bias
=1-127
e=-126

Minimum Possible value of E=00000001, E=1

E= e + bias

e= E – bias, 1-127 =126, so e= -126

That means in worst case e= -126, if value of e is smaller than -126, then number is not able to normalize & we store as Denormalize number.

eg 1.1001×2^{-127} e=-127,E=e+bias, -127+127 = 0 so E = 0,not able to normalize

eg 1.1001×2^{-128} e=-128, so E = -1,not able to normalize

eg

or 1.1001×2^{-129}

Here M=1001

e = -129, E= e + bias, -129+127 = -2, E=-2

So E=-2 not able to normalize because E must be 1 After biasing.

Single Precision :

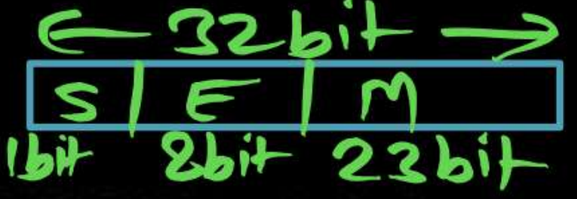
for a Normalized Number.

in Worst Case $e = -126$.

$1.M \times 2^{-126}$ then its Normalized.

otherwise Denormalized Number.

Denormalized Number



Minimum Possible value of E=00000001, E=1

E= e + bias

e= E – bias, 1-127 =126, so e= -126

That means in worst case e= -126, if value of e is smaller than -126, then number is not able to normalize & we store as Denormalize number.

For eg. The Smallest Positive(+ve) single precision Normalized Number is: $1.0000\ 0000\ 0000\ 0000\ 0000\ 000 \times 2^{-126}$ (23bit)

But the Smallest single precision denormalized number is:

$0.0000\ 0000\ 0000\ 0000\ 0000\ 001 \times 2^{-126}$ (23bits (left biasing)) Or 1.0×2^{-149} (M=52bit)

& Double precision Range is: 1.0×2^{-1022} To 1.0×2^{-1074}

Q. How to represent +(1.0) into IEEE 754 single precision floating Point Repartition?

bias = 127

Sol. To represent +(1.0) into IEEE 754 single precision floating Point Repartition..

$+ 1.0 \times 2^0$

$M = 0$

$e = 0$

$E = 127$

S	E	M
1	8	23

$Bias = 2^{8-1}-1 = 127$

$E = e + bias$

$E = 0 + 127$

0	0 1111111	0000 0000 0000 ....

3 F 8 0 0

(3F80 0000)

Q.

Consider a 64 bit Register which store floating point Number in IEEE 754 Double Precision Format.
What is the value of the Number if 64 bit are given as

0 111 1111 1111 0000 0000 0000000000....

Sign = 0 (+ve)

E = 11111111111 = 2047

M = 0

bias = 1023

Sol.

Consider a 64 bit Register which store floating point Number in IEEE 754 Double Precision Format.

What is the value of the Number if 64 bit are given as
0111 1111 1111 0000 0000 0000000000....

S(1 bit)	E(11 bit)	M(52 bit)
0	1111 1111 111	0000000000000

P = (x)
Q = (y)

Here all exponent bits is 1

E = 2047 →
- Infinity (when M = 0)
- Not a number (when M ≠ 0)

Here M = 0 so its $+\infty$

MCQ

The value of a *float* type variable is represented using the single-precision 32-bit floating point format of IEEE-754 standard that uses 1 bit for sign, 8 bits for biased exponent and 23 bits for mantissa. A *float* type variable X is assigned the decimal value of −14.25. The representation of X in hexadecimal notation is

[GATE-2014-Set2-CS: 2M]

A C1640000H

B 416C0000H

C 41640000H

D C16C0000H

-14.25

-1110.01

$1.11001 \times 2^{+3}$

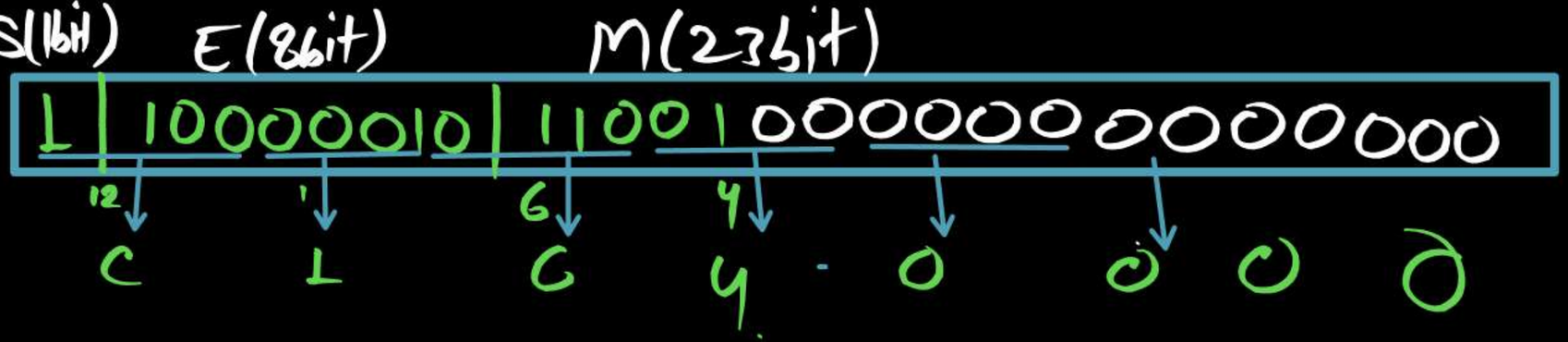
$S = 1$

$E = 130$

$M : 11001$

Ans (A)

$E = e + bias = +3 + 127 = 130$



MCQ

Consider the IEEE-754 single precision floating point numbers
P = 0xC1800000 and Q = 0x3F5C2EF4.
Which one of the following corresponds to the product of these numbers (i.e., P × Q), represented in the IEEE-754 single precision format?

[GATE-2023-CS: 2M]

P = C1800000

Q = 3F5C2EF4

A 0x404C2EF4

B 0x405C2EF4

C 0xC15C2EF4

D 0xC14C2EF4

Denormalized Number

Minimum Possible value of E=00000001, E=1

E= e + bias

e= E – bias, 1-127 =126, so e= -126

That means in worst case e= -126, if value of e is smaller than -126, then number is not able to normalize & we store as Denormalize number.

For eg. The Smallest Positive(+ve) single precision Normalized Number is: $1.0000\ 0000\ 0000\ 0000\ 0000\ 000 \times 2^{-126}$

But the Smallest single precision denormalized number is:

$0.0000\ 0000\ 0000\ 0000\ 0000\ 001 \times 2^{-126}$ Or 1.0×2^{-149}

& Double precision Range is: 1.0×2^{-1022} To 1.0×2^{-1074}

You get

Floating Point Addition

Subtraction

Multiplication

Add/Subtract Rule

1. Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.
2. Set the exponent of the result equal to the larger exponent.
3. Perform addition/subtraction on the mantissas and determine the sign of the result.
   Normalize the resulting value, if necessary.

$1.\ldots\ldots \times 2^{e}$

Multiplication and division are somewhat easier than addition and subtraction, in that no alignment of mantissas is needed.

Multiply Rule

1. Add the exponents and subtract 127.
2. Multiply the mantissas and determine the sign of the result.
3. Normalize the resulting value, if necessary.

→ 1. Something $\times 2^e$

Divide Rule

1. Subtract the exponents and add 127.
2. Divide the mantissas and determine the sign of the result.
3. Normalize the resulting value, if necessary.

The addition or subtraction of 127 in the multiply and divide rules results from using the excess –127 notation for exponents.

Floating-Point Representation

Example:

Show the IEEE 754 binary representation of the number -0.75_{ten} in single and double precision.

Answer:

The number -0.75_{ten} is also.

$-3/4_{ten}$ or $-3/2^2{}_{ten}$

It is also represented by the binary fraction.

$-0.11_{two}/2^2{}_{ten}$ or -0.11_{two}

In scientific notation, the value is.

$-0.11_{two} \times 2^0$

$-(0.75)$ in single precision

-0.75

$\Rightarrow -0.11$ → -0.11×2^0

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

$\Rightarrow \boxed{1.1 \times 2^{-1}}$

$e = -1$

bias $= 127$

$E = e + bias$

$E = 126$

Sign = 1

M = 1000000

$\boxed{E \Rightarrow 01111110}$

1bit	E(8bit)	23bit
1	01111110	10000000000000

And in normalized scientific notation, it is.

$$-1.1_{two} \times 2^{-1}$$

$1.1 = 1.5$

2^0 2^{-1}

$1 + 0.5$

The general representation for a single precision number is.

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

When we subtract the bias 127 from the exponent of $-1.1_{two} \times 2^{-1}$, the result.

$$(-1)^S \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 000_{two}) \times 2^{(126 - 127)}$$

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

S E M

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
1	0	1	1	1	1	1	1	0	1	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

Floating-Point Addition

Let's add numbers in scientific notation by hand to illustrate the problems in floating-point addition:

$$9.999_{ten} \times 10^{1} + 1.610_{ten} \times 10^{-1}.$$

Assume that we can store only four decimal digits of the significand and two decimal digits of the exponent.

Step 1. To be able to add these numbers properly, we must align the decimal point of the number that has the smaller exponent. Hence, we need a form of the smaller number, $1.610_{ten} \times 10^{-1}$, that matches the larger exponent.

We obtain this by observing that there are multiple representations of an unnormalized floating-point number in scientific notation:

$$1.610_{ten} \times 10^{-1} = 0.1610_{ten} \times 10^{0} = 0.01610_{ten} \times 10^{1}$$

$9.999 \times 10^{+1}$

$0.01610 \times 10^{+1}$

Smallest

1.610×10^{-1}

$\Downarrow$

$0.1610 \times 10^{1} \times 10^{-1} \Rightarrow 0.1610 \times 10^{0}$

$0.1610 \times 10^{0} \Rightarrow 0.01610 \times 10^{+1}$

Given only Total 4 Digit Required

9.9999	9.999
0.0016	0.0016
10.0160	10.015

⇒

Round of ⇒ 10.015

The number on the right is the version we desire, since its exponent matches the exponent of the larger number, $9.999_{ten} \times 10^1$. Thus, the first step shifts the significand of the smaller number to the right until its corrected exponent matches that of the larger number. But we can represent only four decimal digits so, after shifting, the number is really:

$$0.016_{ten} \times 10^1$$

Step 2. Next comes the addition of the significands:

$$\begin{array}{r} 9.999_{ten} \\ + \ 0.016_{ten} \\ \hline 10.015_{ten} \end{array}$$

The sum is $10.015_{ten} \times 10^1$.

10.015×10^1

But Normalized Mantissa

$1.xxx$

Step 3.

:This sum is not in normalized scientific notation, so we need to adjust it

$$10.015_{ten} \times 10^{1} = 1.0015_{ten} \times 10^{2}$$

Thus, after the addition we may have to shift the sum to put it into normalized form, adjusting the exponent appropriately. This example shows shifting to the right, but if one number were positive and the other were negative, it would be possible for the sum to have many leading 0s, requiring left shifts. Whenever the exponent is increased or decreased, we must check for overflow or underflow—that is, we must make sure that the exponent still fits in its field.

Step 4.

Since we assumed that the significand can be only four digits long (excluding the sign), we must round the number. In our grammar school algorithm, the rules truncate the number if the digit to the right of the desired point is between 0 and 4 and add 1 to the digit if the number to the right is between 5 and 9. The number

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

Is rounded to four digits in the significant to

$$1.002_{ten} \times 10^2$$

since the fourth digit to the right of the decimal point was between 5 and 9. Notice that if we have bad luck on rounding, such as adding 1 to a string of 9s, the sum may no longer be normalized, and we would need to perform step 3 again.

Steps 1 and 2 are similar to the example just discussed: adjust the significand of the number with the smaller exponent and then add the two significands. Step 3 normalizes the results, forcing a check for overflow or underflow. The test for overflow and underflow in step 3 depends on the precision of the operands. Recall that the pattern of all zero bits in the exponent is reserved and used for the floating-point representation of zero. Also, the pattern of all one bits in the exponent is reserved for indicating values and situations outside the scope of normal floating-point numbers . Thus, for single precision, the maximum exponent is 127, and the minimum exponent is –126. The limits for double precision are 1023 and –1022.

Decimal Floating-Point Addition

Example:

Try adding the numbers 0.5_{ten} and -0.4375_{ten} in binary using the algorithm in Figure:

Answer:

Let's first look at the binary version of the two numbers in normalized scientific notation, assuming that we keep 4 bits of precision:

$$0.5_{ten} = 1/2_{ten} = 1/2^{1}_{ten}$$

$$= 0.1_{two} = 0.1_{two} \times 2^{0} = 1.000_{two} \times 2^{-1}$$

$$-0.4375_{ten} = -7/16_{ten} = -7/2^{4}_{ten}$$

$$= -0.0111_{two} = -0.0111_{two} \times 2^{0} = -1.110_{two} \times 2^{-2}$$

0.5 0.4375

Now we follow the algorithm:

Step 1. The significand of the number with the lesser exponent ($-1.11_{two} \times 2^{-2}$) is shifted right until its exponent matches the larger number:

$-1.110_{two} \times 2^{-2} = -0.111_{two} \times 2^{-1}$ & 1.000×2^{-1}

Step 2. Add the significands:

$$1.000_{two} \times 2^{-1} + (-0.111_{two} \times 2^{-1}) = 0.001_{two} \times 2^{-1}$$

Now Normalize 1. something

Step 3. Normalization the sum, checking for overflow or underflow:

e = −4

$$0.001_{two} \times 2^{-1} = 0.010_{two} \times 2^{-2} = 0.100_{two} \times 2^{-3}$$
$$= 1.000_{two} \times 2^{-4}$$ Normalized Mantissa

Since 127 ≥ −4 ≥ − 126, there is no overflow or underflow. (The biased exponent would be −4 + 127, or 123, which is between 1 and 254, the smallest and largest unreserved biased exponents.

Step 4. Round the sum:

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

The sum already fits exactly in 4 bits, so there is no change to the bits due to rounding.

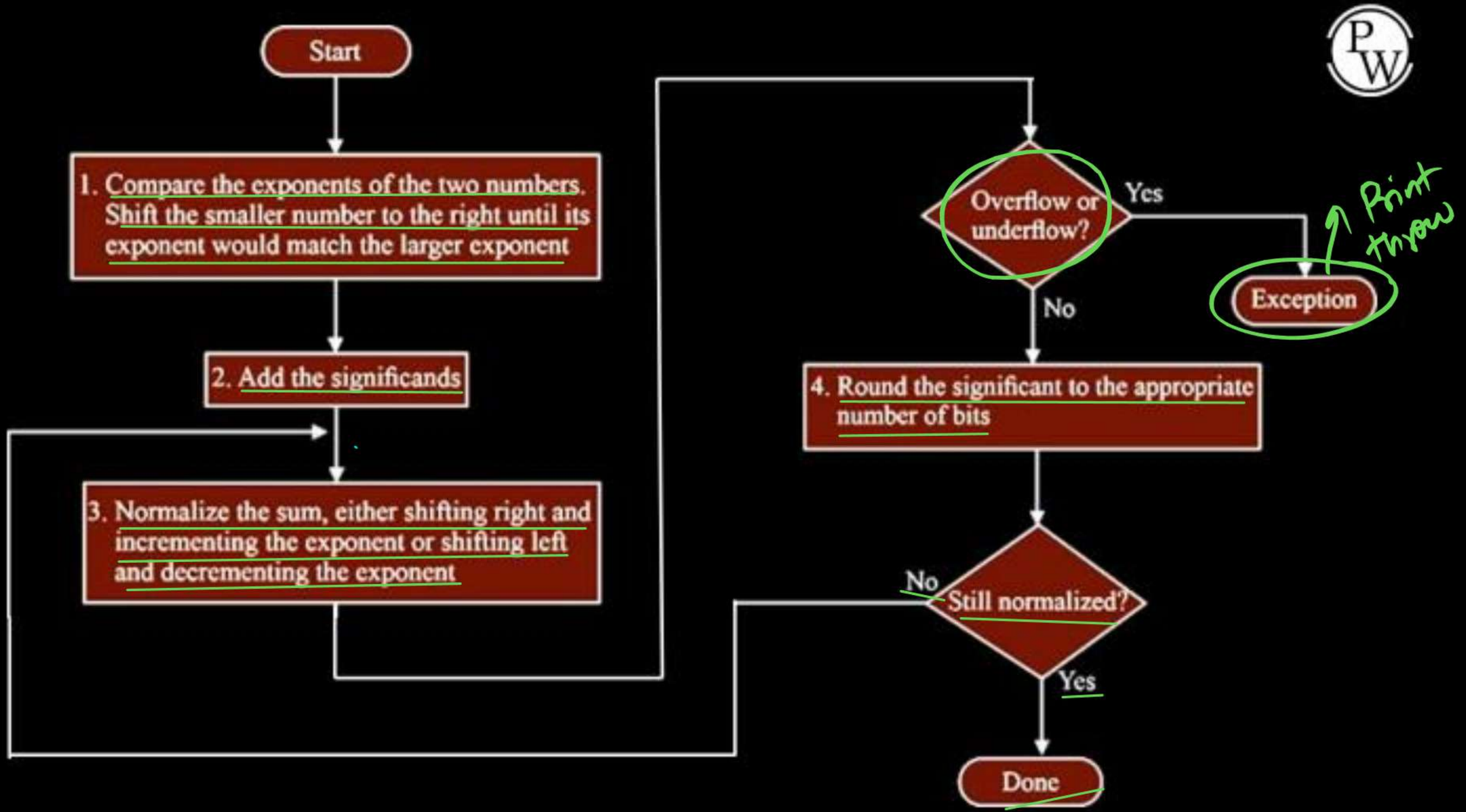
This sum is then

$$1.000_{two} \times 2^{-4} = 0.0001000_{two} = 0.0001_{two}$$

$$= 1/2^{4}_{ten} = 1/16_{ten} = 0.0625_{ten}$$

This sum is what we would expect from adding 0.5_{ten} to -0.4375_{ten}.

Many computers dedicate hardware to run floating-point operations as fast as possible. Figure sketches the basic organization of hardware for floating-point addition.


Start
1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
Print throw
No
4. Round the significant to the appropriate number of bits
No
Still normalized?
Yes
Done

Floating-point addition. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be abnormalized, we must repeat step 3.

Floating Point Multiplication.

GATE - 2023 — 2 Marks Question Asked.

Floating-Point Multiplication

Now that we have explained floating-point addition, let's try floating-point multiplication. We start by multiplying decimal numbers in scientific notation by hand: $1.110_{ten} \times 10^{10} \times 9.200_{ten} \times 10^{-5}$. Assume that we can store only four digits of the significand and two digits of the exponent. Step 1. Unlike addition, we calculate the exponent of the product by simply adding the exponents of the operands together:

$P = 1.110 \times 10^{10}$ & $Q = 9.200 \times 10^{-5}$

Step1 Directly add the exponent.

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

$e = 5$

$E = e + bias = 5 + 127$

$E = 132$

$e = 10 \quad e = -5$

$bias = 127$

$e = -5$

$10 + (-5) = 5$

$e = 5$

Cross checking

$e = 10$

$E = e + bias = 10 + 127$

$E = 137$

$E = -5 + 127$

$E = 122$

$137 + 122 = 259 - 127$

$= 132 \Rightarrow e = 5$

$bias = 127$

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

New exponent = 10 + (–5) = 5

Let's do this with the biased exponents as well to make sure we obtain the same result:

Only CROSS checking method

10 + 127 = 137, and –5 + 127 = 122, so

New exponent = 137 + 122 = 259

This result is too large for the 8-bit exponent field, so something is amiss! The problem is with the bias because we are adding the biases as well as the exponents:

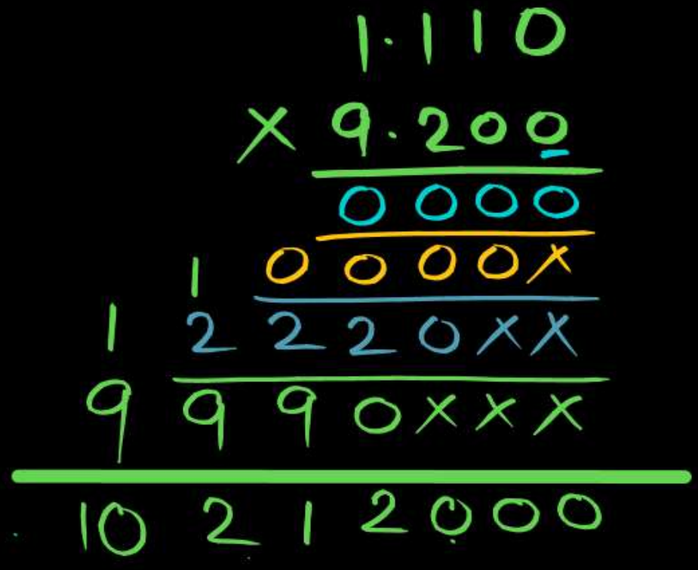
New exponent = (10 + 127) + (–5 +127) = (5 + 2 × 127) = 259

Accordingly, to get the correct biased sum when we add biased numbers, we must subtract the bias from the sum:

New exponent = 137 + 122 – 127 = 259 – 127 = 132 = (5 + 127) and 5 is indeed the exponent we calculated initially.

Step 2.Next comes the multiplication of the significands:

$$
\begin{array}{r}
1.110_{ten} \\
\times\ \underline{9.200_{ten}} \\
0000 \\
\hline
0000\,x \\
\hline
2220 \\
\hline
9990 \\
\hline
10212000_{ten}
\end{array}
$$



There are three digits to the right of the decimal for each operand, so the decimal point is placed six digits from the right in the product significand:

$$10.212000_{ten}$$

Assuming that we can keep only three digits to the right of the decimal point, the product is $\boxed{10.212 \times 10^5}$.

Step 3. This product is unnormalized, so we need to normalize it:

Normalized

$$10.212_{ten} \times 10^5 = 1.0212_{ten} \times 10^6$$

Thus, after multiplication, the product can be shifted right one digit to put it in normalized form, adding 1 to the exponent. At this point, we can check for overflow and underflow. Underflow may occur if both operands are small—that is, if both have large negative exponents.

Step 4. We assumed that the significand is only four digits long (excluding the sign), so we must round the number.

The number $1.0212_{ten} \times 10^6$

is rounded to four digits in the significand to

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

Step 5. The sign of the product depends on the signs of the original operands. they are both the same, the sign is positive; otherwise, it's negative. Hence the product is

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

The sign of the sum in the addition algorithm was determined by addition of the significands, but in multiplication the sign of the product is determined by the signs of the operands.

Once again, as Figure shows, multiplication of binary floating-point numbers is quite similar to the steps we have just completed. We start with calculating the new exponent of the product by adding the biased exponents, being sure to subtract one bias to get the proper result. Next is multiplication of significands, followed by an optional normalization step. The size of the exponent is checked for overflow or underflow, and then the product is rounded. If rounding leads to further normalization, we once again check for exponent size. Finally, set the sign bit to 1 if the signs of the operands were different (negative product) or to 0 if they were the same (positive product).

p & q.

1. Directly Add the exponent.
2. Multiply the Fractional Part.
3. Normalize the Mantissa.

Decimal Floating-Point Multiplication

P = 0.5

0.1×2^{0}

1.0×2^{-1}

Q = −0.4375

-0.0111×2^{0}

1.11×2^{-2}

e = −1

e = −2

Example:

Let's try multiplying the numbers 0.5_{ten} and -0.4375_{ten}, using the steps in Figure.

Answer:

In binary, the task is multiplying $1.000_{two} \times 2^{-1}$ by $-1.110_{two} \times 2^{-2}$.

Step 1. Adding the exponents without bias:

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

or, using the biased representation:

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

E = −3 + 127

E = 124

Step 2. Multiplying the significands:

$$
\begin{array}{r}
1.000_{two} \\
\times\ 1.110_{two} \\
\hline
0000 \\
1000\times \\
1000\times\times \\
1000\times\times\times \\
\hline
1110000_{two}
\end{array}
$$

e = -3

The product is $1.110000_{two} \times 2^{-3}$, but we need to keep it to 4 bits, so it is $1.110_{two} \times 2^{-3}$.

Step 3. Now we check the product to make sure it is normalized, and then check the exponent for overflow or underflow. The product is already normalized and, since $127 \geq -3 \geq -126$, there is no overflow or underflow. (Using the biased representation, $254 \geq 124 \geq 1$, so the exponent fits.)

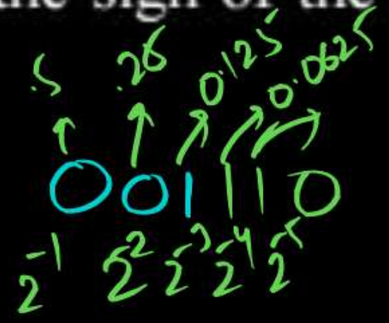
Step 4. Rounding the product makes no change:

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

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

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

Ans

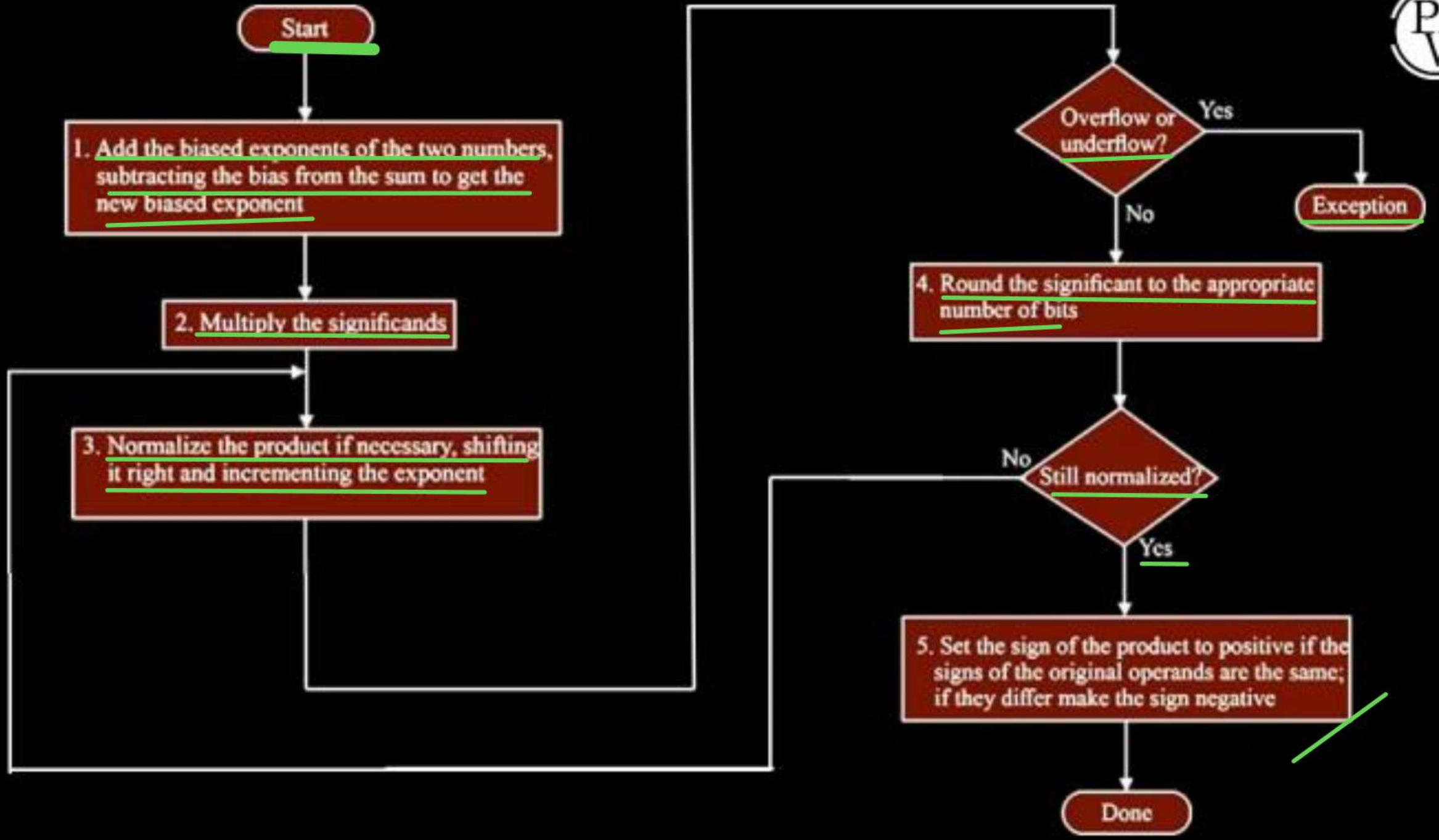


Converting to decimal to check our results:

$$-1.110_{two} \times 2^{-3} = -0.001110_{two} = -0.00111_{two}$$

$$= -7/2^{5}{}_{ten} = -7/32_{ten} = -0.21875_{ten}$$

The product of 0.5_{ten} and -0.4375_{ten} is indeed -0.21875_{ten}.


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 significant to the appropriate number of bits
No
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

Floating multiplication:

The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

Arithmetic Operations on Floating-Point Number

In this section, we outing the general procedure for addition, subtraction, multiplication, and division of floating-point numbers. The rules we give apply to the single-precision IEEE standard format. These rules specify only the major steps needed to perform the four operations; for example, the possibility that overflow or underflow might occur is not discussed. Furthermore, intermediate results for both mantissas and exponents might require more than 24 and 8 bits, respectively. These and other aspects of the operations must be carefully considered in designing an arithmetic unit that meets the standard. Although we do not provide full details in specifying the rules, we consider some aspects of implementation, including rounding, in later sections.

If their exponents differ, the mantissas of floating-point numbers must be shifted with respect to each other before they are added or subtracted. Consider a decimal example in which we wish to add 2.9400×10^2 to 4.3100×10^4. We rewrite 2.9400×10^2 as 0.0294×10^4 and then perform additional of the mantissas to get 4.3394×10^4. The rule for addition and subtraction can be stated as follows:

Add/Subtract Rule

1. Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.
2. Set the exponent of the result equal to the larger exponent.
3. Perform addition/subtraction on the mantissas and determine the sign of the result.

   Normalize the resulting value, if necessary.

Multiplication and division are somewhat easier than addition and subtraction, in that no alignment of mantissas is needed.

Multiply Rule

1. Add the exponents and subtract 127.
2. Multiply the mantissas and determine the sign of the result.
3. Normalize the resulting value, if necessary.

Divide Rule

1. Subtract the exponents and add 127.
2. Divide the mantissas and determine the sign of the result.
3. Normalize the resulting value, if necessary.

The addition or subtraction of 127 in the multiply and divide rules results from using the excess –127 notation for exponents.

Type of Instruction / Operation

1. DATA Transfer Instⁿ / operation
2. Data Manipulation Instⁿ
   - (i) Airthmatic (ADD, SUB, DIV, MUL, INC etc)
   - (ii) Logical (OR, NOT, AND, XOR etc)
   - (iii) Shift & Rotate operation
3. Program Control (TOC / Transfer of Control) Instⁿ

① Data Transfer Instn

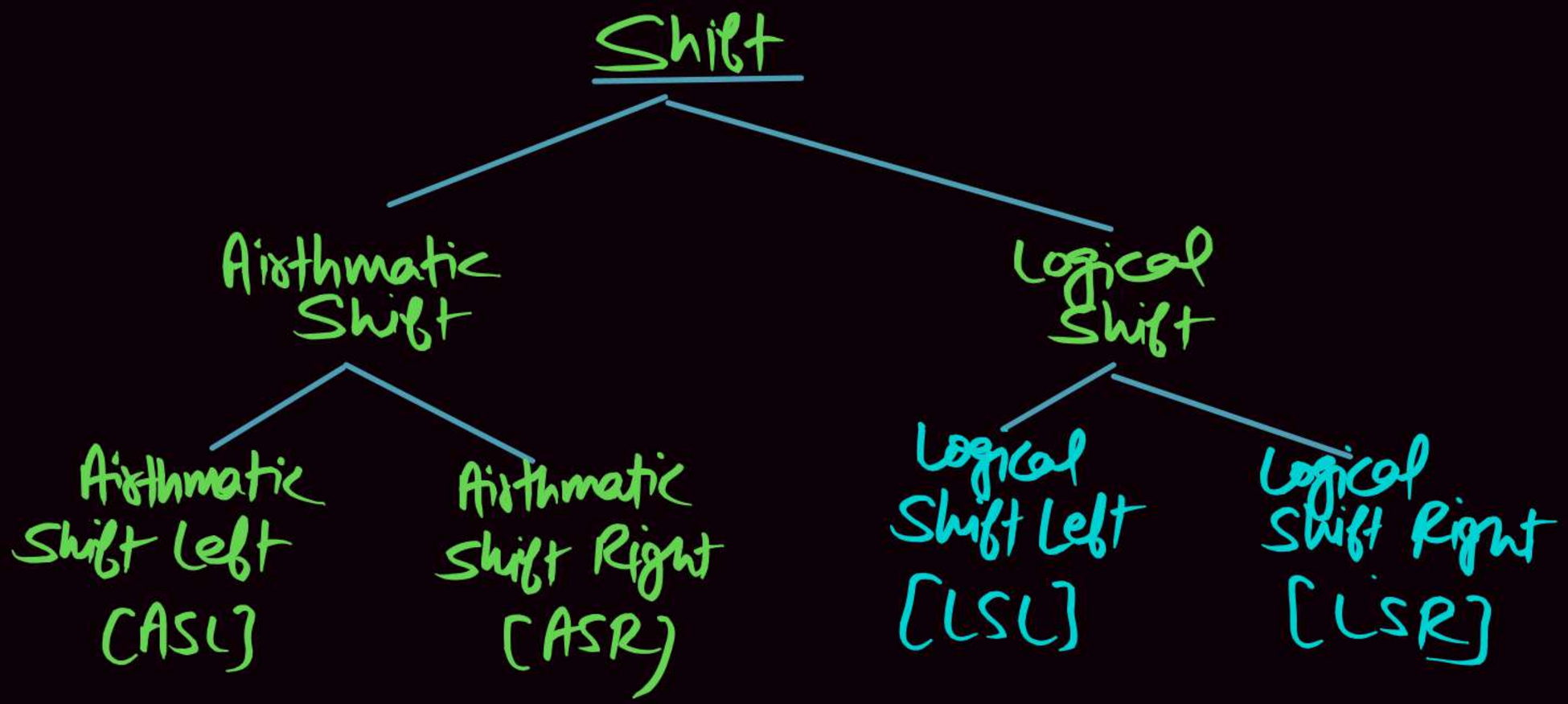
'MOV' → General Data Transfer operⁿ

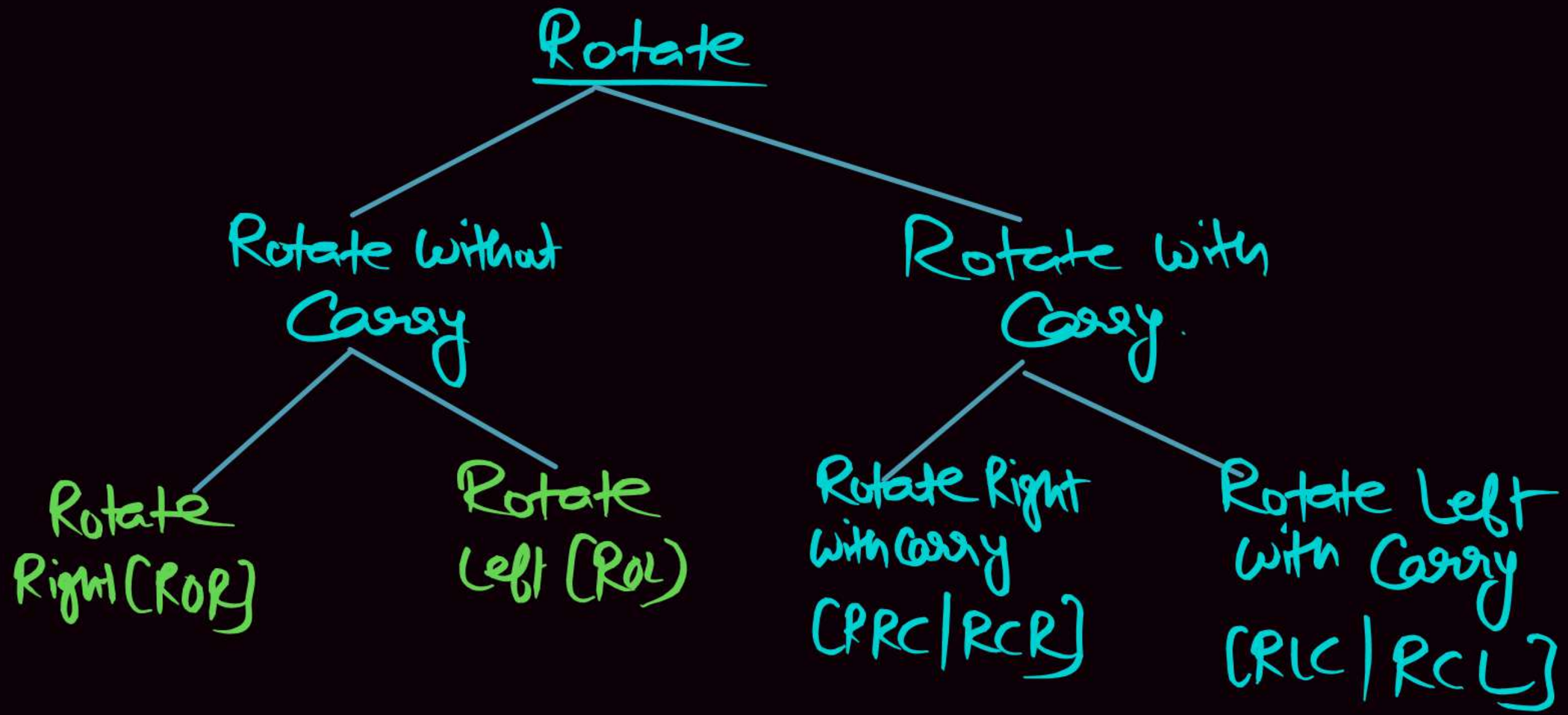
Destⁿ	Source
Reg	Reg
Reg	Mem
Mem	Reg
Reg	Immediate AM
Mem	Immediate AM

- LOAD, STORE → memory Specific Data transfer
  - LOAD → Memory Read
  - STORE → memory write
- PUSH, POP → Stack Specific
  - PUSH → Insert
  - POP → Retrive/Delete
- IN, OUT → Input output Specific
  - IN → I/o Read
  - OUT → I/o write

Immediate → Constant (No Storage)

SHIFT & Rotate operation.




Rotate
Rotate without Carry
Rotate with Carry
Rotate Right [ROR]
Rotate Left [ROL]
Rotate Right with Carry [RRC | RCR]
Rotate Left with Carry [RLC | RCL]

Program Control
(Transfer of Control)
(TOC) Instn

Unconditional TOC

(eg) JMP x
HALT

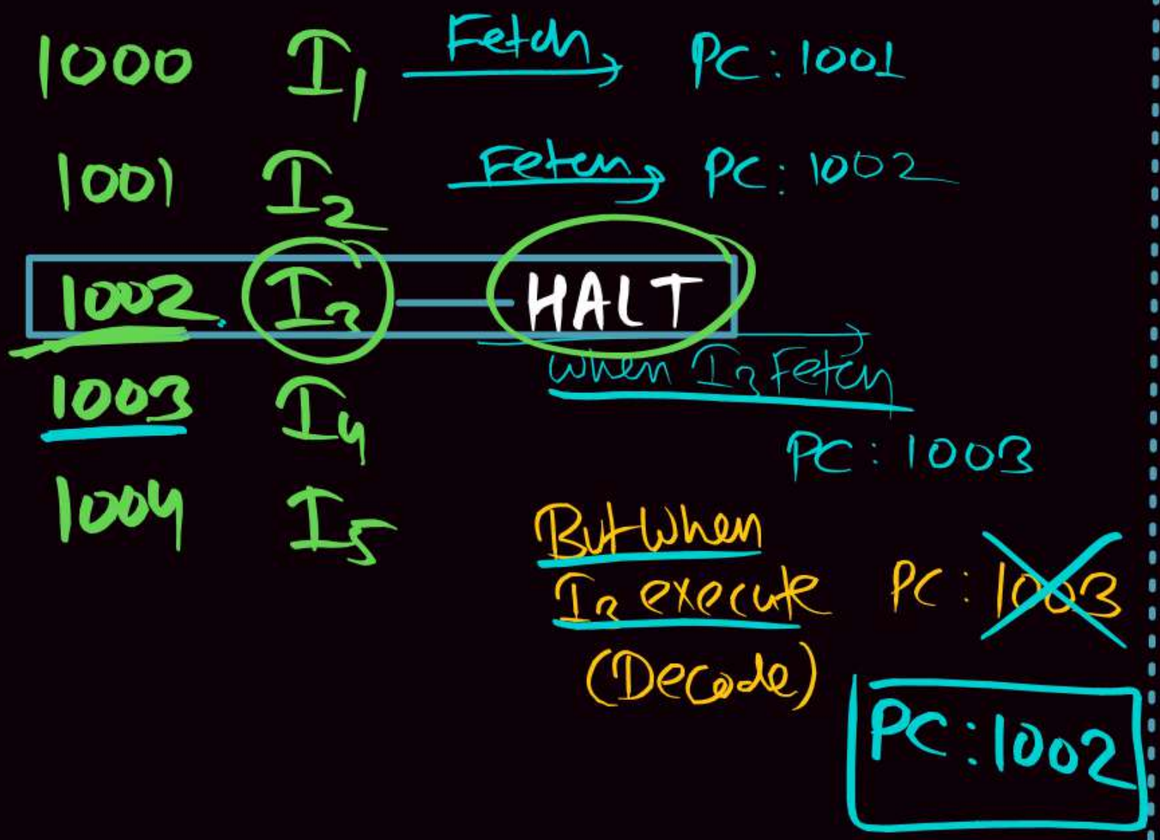
x → Target Addr. already Taken in ALU

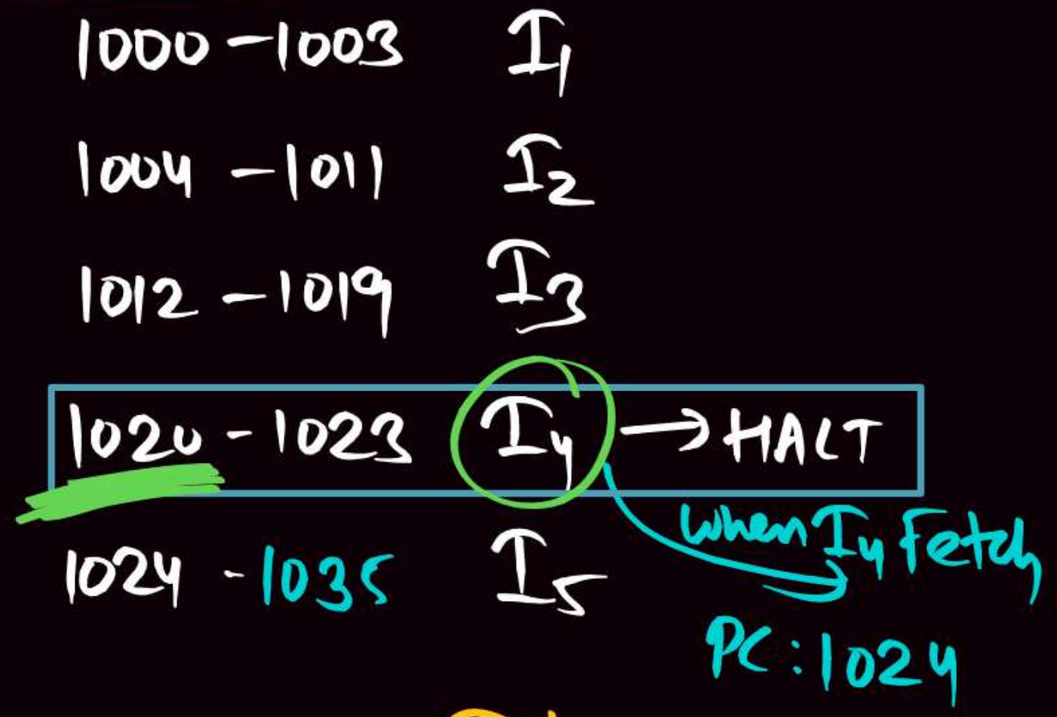
Conditional TOC

(eg) JNZ (Jump on Not zero)
JZ
etc.

I_x : HALT ⇒ JMP I_x

HALT: its Un Condition Jmp Instn in which target address is the Starting address of Instn which called HALT Instn





Q. Consider the following program segment for a hypothetical CPU Having three users registers R1, R2 and R3.

[GATE-2 Marks]

	Instruction	Operation	Instruction size (in words)
1000 - 1007	MOV R1, 5000 I_1	R1 ← Memory[5000]	2
1008 - 1011	MOVR2, (R1) I_2	R2 ← Memory[(R1)]	1
1012 - 1015	ADD R2, R3 I_3	R2 ← R2 + R3	1
1016 - 1023	MOV 6000, R2 I_4	Memory [6000] ← R2	2
1024 - 1027	HALT I_5	Machine Halts	1

→ when I_5 Feten then PC: 1028
But when Halt execute

1028 —

Consider that the memory is word addressable with size 32 bits and the program has been loaded starting from memory location 1000 (decimal). If an interrupt occurs while the CPU has been halted after executing the HALT instruction, the return address (in decimal) saved in the stack will be

PC: ~~1026~~ 1024 Ans

(a) 1007 (b) 1020 (c) 1024 (d) 1028

THANK YOU!