UNIT 3

DATA REPRESENTATION AND COMPUTER ARITHMETIC

PART-A
SHORT QUESTIONS WITH SOLUTIONS

Q1. Write about decimal number system.

Answer : Model Paper-II, Q1(e)

Decimal Number System

The decimal number system employs the base 10 system and the ten different symbols used in this system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. A string in decimal number system is distinguished from other number systems by enclosing it in parenthesis and inserting base 10 as its subscript, for example, $(4562.95)_{10}$.

Example

The string $(4562.95)_{10}$ is interpreted as, $4 \times 10^3 + 5 \times 10^2 + 6 \times 10^1 + 2 \times 10^0 + 9 \times 10^{-1} + 5 \times 10^{-2}$

$$= 4 \times 1000 + 5 \times 100 + 6 \times 10 + 2 \times 1 + 9 \times \frac{1}{10} + 5 \times \frac{1}{100}$$

$$= 4000 + 500 + 60 + 2 + 0.9 + 0.05$$

$$= (4562.95)_{10}$$

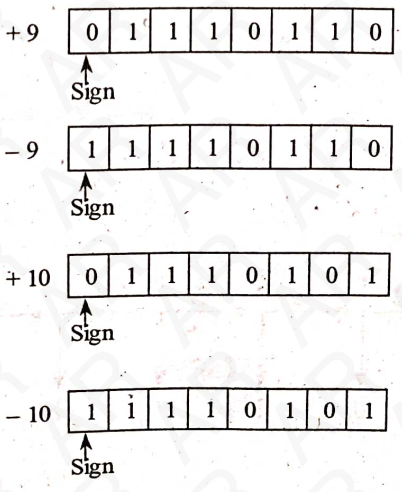
Q2. Discuss in short about signed 1's complement representation.

Answer :

1's complement representation was brought up to replace the sign magnitude representation. But, it did not serve longer. Many shortcomings overwhelmed even with this representation.

In order to represent a given signed number in terms of 1's complement representation, the magnitude of the number should be converted from 1 to 0 and 0 to 1. There will be no change in case of representation of sign of the number (i.e.,) in order to represent negative sign a value '1' is used at most significant bit position and a value '0' to represent the positive sign of the number.

For example consider the representation of decimal 9 and 10 in terms of 1's complement.



Q3. What is 10's complement?

Answer : Model Paper-III, Q1(e)

The 10's complement of a decimal number N is defined as,

$$\boxed{10^n - N}$$

Where, $10^n = 1$ followed by n 0s

In other words, the 10's complement of N can be obtained by keeping all least significant zeroes unchanged, subtracting the first non-zero least significant digit from 10 and then subtracting all the remaining higher significant digits from 9.

Example

The 10's complement of 25783 is obtained by subtracting the first non-zero least significant digit, i.e., 3 from 10 and then subtracting the remaining higher order digits i.e., 8, 7, 5 and 2 from 9. Thus 10's complement of 25783 is 74217.

An alternative method of finding 10's complement of N is adding 1 to its 9's complement.

The 9's complement of 25783 is 74216. Thus, the 10's complement of 25783 is 74216 + 1 = 74217.

Q4. Write short notes on floating-point representation of decimal number.

Answer :

As mentioned earlier, the floating-point number is basically divided into two segments, with one segment representing the mantissa and other representing the exponent. Hence it takes the form as,

$$M \times K^e$$

Where 'M' corresponds to mantissa, 'e' corresponds to exponent and 'K' corresponds to radix.

For example, consider two floating-point numbers i.e., + 122.28 and + 5089.34

The floating-point number, i.e., 122.28 can be written as $+ 0.12228 \times 10^{+3}$ ($M \times K^e$). Where + 0.12228 forms the fractional part or mantissa, the power '3' is exponent and '10' corresponds to radix. Similarly +5089.34 can be written as $+ 0.508934 \times 10^{+4}$ ($M \times K^e$), where 0.5089 corresponds to fractional or mantissa part, + 4 corresponds to the exponent and '10' is the radix. While dealing with floating-point representation of decimal numbers one has to remember that the computer accommodates only the mantissa and exponents part including their signs in register while radix is just assumed.

Q5. What is the gray code equivalent of the Hex Number 3A7?

Answer : Model Paper-I, Q1(e)

Given hexadecimal number = $(3A7)_{16}$

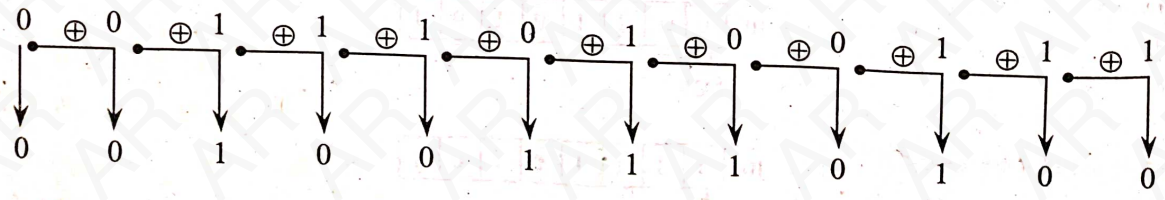
Converting the hexadecimal number to binary,

We get,

3	A	7
0011	1010	0111

$\therefore \quad (3A7)_{16} = (001110100111)_2$

Its gray code equivalent is computed as,



$\therefore \quad (3A7)_{16} = (001001110100)_G$

Q6. Find 9's complement $(25.639)_{10}$.

Answer :

We know that,

9's complement of number, $N = (10^n - 1) - N$

$\therefore$ 9's complement of $(25.639)_{10}$

$= (100.000 - 1) - 25.639$

$= 99.999 - 25.639$

$= 74.360$

Q7. Define fast multiplication circuit.

Answer :

Fast multiplication circuits are used when long multiplicand is given, (of size 1024 or 2048 bits or above). Performing bit-by-bit shifting operation on such a long multiplicand is computationally expensive.

Thus, to speed up the multiplication process fast multiplication circuits are used that provide fast adder structures in order to achieve optimal performance of n-bit by n-bit multiplication.

Q8. Obtain divide operation on $\frac{110}{111}$.

Answer : Model Paper-II, Q1(f)

Dividend: 110

Divisor: 111

```
  0. 1 1 0 (quotient)
111)1 1 0 0
    1 1 1
  ---------
    1 0 1 0
      1 1 1
  ---------
    0 1 1 0
    0 0 0 0
  ---------
      1 1 0 (Remainder)
  ---------
```

$\therefore 110/111 = 0.\overline{110}$

Q9. Write short notes on underflows.

Answer : Model Paper-III, Q1(f)

An underflow occurs while performing subtraction operation i.e. whenever there is a borrow beyond most significant bit, we say that an underflow occurred. Whenever underflow is observed, it means that the result obtained is too small to be represented. Let us analyze this consequence by adhering to a simple example.

```
Indicates underflow ---->   1 1
                A :     1 0 0 1
                B :     1 0 1 0
            A - B :     0 1 1 1
```

Obviously if the computer is considered as infinite system there will not be any problems related to overflows since, the memory will be sufficient enough to manage overflow bits resulting out of computer arithmetic. But it is impractical to analyze such systems since the systems may get subjected to many unpredictable problems.

Q10. Write short notes on overflows.

Answer : Model Paper-I, Q1(f)

During certain addition operations, it may happen that the carry may occur beyond MSB (Most Significant Bit) position. Such situations are often referred to as overflow condition or situation. This concept should be considered, since, every computer possesses fixed sized registers and any carry beyond this cannot be accommodated. Hence, in order to satisfy this condition, computer uses special circuitry which is referred as flip-flop to hold the overflow bit.

For example, consider the addition between – 70 and – 80.

```
  -70  1 0 1 1 1 0 1 0
  -80  1 0 1 1 0 0 0 0
      ---------------
Overflow bit -> (1) 0 1 1 0 1 0 1 0
```

PART-B
ESSAY QUESTIONS WITH SOLUTIONS

3.1 DATA REPRESENTATION

3.1.1 Data Types

Q11. Define each of the following number systems,

(i) Decimal

(ii) Binary

(iii) Octal

(iv) Hexadecimal.

Answer : Model Paper-II, Q6(a)

(i) Decimal Number System

The decimal number system employs the base 10 system and the ten different symbols used in this system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. A string in decimal number system is distinguished from other number systems by enclosing it in parenthesis and inserting base 10 as its subscript, for example, $(4562.95)_{10}$.

Example

The string $(4562.95)_{10}$ is interpreted as,

$$4 \times 10^3 + 5 \times 10^2 + 6 \times 10^1 + 2 \times 10^0 + 9 \times 10^{-1} + 5 \times 10^{-2}$$

$$= 4 \times 1000 + 5 \times 100 + 6 \times 10 + 2 \times 1 + 9 \times \frac{1}{10} + 5 \times \frac{1}{100}$$

$$= 4000 + 500 + 60 + 2 + 0.9 + 0.05$$

$$= (4562.95)_{10}$$

(ii) Binary Number System

The binary number system employs base 2 system and the two different symbols used in this system are 0 and 1. A string in binary number system is distinguished from other number systems by enclosing it in parenthesis and inserting base 2 as its subscript, for example $(110110)_2$.

Example

The string $(110110)_2$ is interpreted as,

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

$$= 1 \times 32 + 1 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1$$

$$= 32 + 16 + 0 + 4 + 2 + 0$$

$$= (54)_{10}$$

Therefore, $(110110)_2 = (54)_{10}$, which means $(110110)_2$ is a string in binary number system and is equivalent to the string $(54)_{10}$ in decimal number system.

(iii) Octal Number System

The octal number system employs base 8 system and the eight different symbols used in this system are 0, 1, 2, 3, 4, 5, 6 and 7. A string in octal number system is distinguished from other number systems by enclosing it in parenthesis and inserting the base 8 as its subscript. For example, $(162.54)_8$.

Example

The string $(162.4)_8$ is interpreted as,

$$1 \times 8^2 + 6 \times 8^1 + 2 \times 8^0 + 4 \times 8^{-1}$$

$$= 1\times64 + 6\times8 + 2\times1 + 4\times\frac{1}{8}$$

$$= 64 + 48 + 2 + \frac{1}{2}$$

$$= 114 + 0.5$$

$$= (114.5)_{10}$$

Therefore, $(162.4)_8 = (114.5)_{10}$, which means the string $(162.4)_8$ in octal number system is equivalent to the string $(114.5)_{10}$ decimal number system.

(iv) Hexadecimal Number System

The hexadecimal number system employs base 16 system and the sixteen different symbols used in this system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, *A*, *B*, *C*, *D*, *E* and *F*. The symbols *A*, *B*, *C*, *D*, *E* and *F* correspond to 10, 11, 12, 13, 14 and 15 respectively in decimal number system. A string in a hexadecimal number system is distinguished from other number systems by enclosing it in parenthesis and inserting the base 16 as its subscript, for example, $(C2B)_{16}$.

Example

The string $(C2B)_{16}$ is interpreted as,

$$C \times 16^2 + 2 \times 16^1 + B \times 16^0$$

$$= 12 \times 256 + 2 \times 16 + 11 \times 1$$

$$= 3072 + 32 + 11$$

$$= (3115)_{10}$$

Therefore, $(C2B)_{16} = (3115)_{10}$, which means the string $(C2B)_{16}$ in hexadecimal number system is equivalent to the string $(3115)_{10}$ in decimal number system.

Q12. Explain how a binary number can be converted to an octal and a hexadecimal number.

Answer :

Binary to Octal Conversion

In this conversion, the binary number is first partitioned into groups of three binary digits, starting from the lower order bit. If the last group of higher order bits is not complete, then zeroes are added as higher order bits to complete the group.

Each group is then assigned to its octal equivalent. Thus, each group corresponds to binary-coded octal numbers. The first eleven binary-coded octal numbers are listed below,

Decimal Equivalent	Octal Number	Binary-coded Octal number
0	0	000
1	1	001
2	2	010
3	3	011
4	4	100
5	5	101
6	6	110
7	7	111
8	10	001 000
9	11	001 001
10	12	001 010

Example

Converting the binary string $(1100010)_2$ into an octal number.

Step 1

Divide the given binary number into groups of three binary digits.

$$(\underline{1}\ \underline{100}\ \underline{010})_2$$

Step 2

Add 0's at high order bit positions to complete the last group.

$$(\underline{001}\ \underline{100}\ \underline{010})_2$$

Step 3

The octal number equivalent of each group is,

$$\begin{array}{ccc} \underline{001} & \underline{100} & \underline{010} \\ 1 & 4 & 2 \end{array} \quad [\because \text{From the above table}]$$

Thus, the octal number equivalent of,

$(1100010)_2$ is $(142)_8$.

Binary to Hexadecimal Conversion

In this conversion, the binary number is first partitioned into groups of four binary digits, starting from lower order bit. If the last group of higher order bits is not complete, then zeroes are added as higher order bits to complete the group.

Each group is then assigned to its hexadecimal equivalent. Thus, each group corresponds to binary-coded hexadecimal numbers. The first twenty binary-coded hexadecimal numbers are listed below,

Decimal Equivalent	Hexadecimal Number	Binary-coded Hexadecimal Number
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	0001 0000
17	11	0001 0001
18	12	0001 0010
19	13	0001 0011

Example

Converting $(1111010111000)_2$ into a hexadecimal number.

Step 1

Divide the given binary string into groups of four binary digits, starting from lower-order bit.

$$\underline{1}\,\underline{1110}\,\underline{1011}\,\underline{1000}$$

Step 2

Add 0's at higher order bit positions to complete the last group.

$$\underline{0001}\,\underline{1110}\,\underline{1011}\,\underline{1000}$$

Step 3

The hexadecimal equivalent of each group is

0001	1110	1011	1000
1	E	B	8

[∴ From the above table]

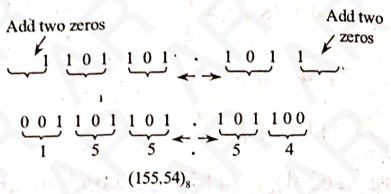
Thus, the hexadecimal equivalent of $(1111010111000)_2$ is $(1EB8)_{16}$

Q13. Convert 1101101.1011 into it's octal, decimal and hexadecimal equivalents.

Answer :

Octal Equivalent

1101101.1011



$(155.54)_8$

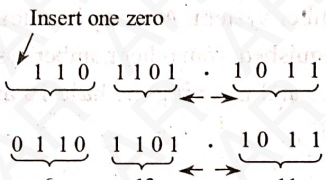
Decimal Equivalent

$1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4}$

$= 64 + 32 + 8 + 4 + 1 + 0.5 + 0.125 + 0.0625$

$= (109.6875)_{10}$

Hexadecimal Equivalent

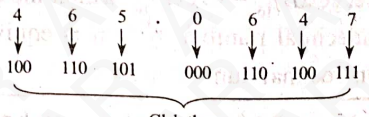


$(6D . B)_{16}$

Q14. Convert $(465.0647)_8$ into binary, decimal and hexadecimal equivalents.

Answer :

Binary Equivalent



$(100\ 110\ 101\ .\ 000\ 110\ 100\ 111)_2$

Decimal Equivalent

$465.0647 = 4 \times 8^2 + 6 \times 8^1 + 5 \times 8^0 + 0 \times 8^{-1} + 6 \times 8^{-2} + 4 \times 8^{-3} + 7 \times 8^{-4}$

$= 256 + 48 + 5 + 0 + 0.094 + 0.0078 + 0.171$

$= (309.2728)_{10}$

Hexadecimal Equivalent

465. 0647

Step 1

Convert the above value into its binary equivalent,

i.e., $(1\ 0\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ .\ 0\ 0\ 0\ 1\ 1\ 0\ 1\ 0\ 0\ 1\ 1\ 1)_2$

Step 2

Convert the above binary data into its hexadecimal equivalent.

Insert three zeros

1 0011 0101 . 0001 1010 0111

0001 0011 0101 . 0001 1010 0111

1 3 5 . 1 10 7

$(135.1A7)_{16}$

Q15. Convert $(19.625)_{10}$ into its binary, octal and hexadecimal equivalents.

Answer : Model Paper-I, Q6(a)

Binary Conversion

19 .625

2	19		
2	9 — 1		
2	4 — 1		
2	2 — 0		
	1		

$(1011)_2$

$0.625 \times 2 = 1.25$ 0.25 1

$0.25 \times 2 = 0.5$ 0.5 0

$0.5 \times 2 = 1.0$ 0.0 1

$(.101)_2$

Combine

$(1011.101)_2$

Octal Conversion

19.625

8 | 19

2 — 3

$(23)_8$

$0.625 \times 8 = 5.0$ 0.0 5

$(0.5)_8$

Combine

$(23.5)_8$

Hexadecimal Conversion

19.625

16 | 19

1 — 3

$(13)_{16}$

$0.625 \times 16 = 10.0$ 0.0 10

$(0.10)_{16}$

Combine

$(13.10)_{16}$

$(13.A)_{16}$

Q16. Convert $(10A4.249)_{16}$ into it's binary, octal and decimal equivalents.

Answer :

Binary Conversion

10A4 . 249

0001 0000 1010 0100 . 0010 0100 1001

As it has no significance, hence it can be omitted.

$(1000010100100.001001001001)_2$

Octal Conversion

10A4.249

Step 1

Convert the above number into its binary equivalent which is nothing but,

1 0000 1010 0100 . 0010 0100 1001

Step 2

Convert the above binary number into its octal equivalent.

Insert two zeros

1 000 010 100 100 . 001 001 001 001

001 000 010 100 100 . 001 001 001 001

1 0 2 4 4 . 1 1 1 1

$(10244.1111)_8$

Decimal Conversion

10A4.249

Step 1

Convert the above number into its binary equivalent which is nothing but,

1 0000 1010 0100 . 0010 0100 1001

Step 2

Convert the above binary number into it's decimal equivalent.

1 0000 1010 0100 . 0010 0100 1001

$1 \times 2^{12} + 0 \times 2^{11} + 0 \times 2^{10} + 0 \times 2^9 + 0 \times 2^8 + 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 0 \times 2^{-5} + 1 \times 2^{-6} + 0 \times 2^{-7} + 0 \times 2^{-8} + 1 \times 2^{-9} + 0 \times 2^{-10} + 0 \times 2^{-11} + 1 \times 2^{-12}$

$= 4096 + 0 + 0 + 0 + 0 + 128 + 0 + 32 + 0 + 0 + 4 + 0 + 0 + 0 + 0 + 0.125 + 0 + 0 + 0.0156 + 0 + 0 + 0.00196 + 0 + 0 + 0.00024$

$=$ Approx $(4260.1428)_{10}$

Q17. Convert the following numbers,

(a) 10101100111.0101 to Base 10

(b) $(153.513)_{10} = (\ )_8$

(c) Given that $(292)_{10} = (1204)_b$ determine 'b'.

Answer :

(a) 10101100111.0101 to Base 10

Given binary number is,

$(10101100111.0101)_2$

Its decimal equivalent is

$$= (1 \times 2^{10}) + (0 \times 2^9) + (1 \times 2^8) + (0 \times 2^7) + (1 \times 2^6) + (1 \times 2^5) + (0 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4})$$

$$= 2^{10} + 2^8 + 2^6 + 2^5 + 2^2 + 2^1 + 2^0 + 2^{-2} + 2^{-4}$$

$$= 1024 + 256 + 64 + 32 + 4 + 2 + 1 + 0.25 + 0.0625$$

$$= (1383.3125)_{10}$$

$\therefore (10101100111.0101)_2 = (1383.3125)_{10}$

(b) $(153.513)_{10} = (\ )_8$

Given decimal number is,

$(153.513)_{10}$

Its octal equivalent is obtained as follows,

8 | 153
8 | 19 – 1 ↑
2 – 3

$\therefore (153)_{10} = (231)_8$

$0.513 \times 8 = 4.104$

$0.104 \times 8 = 0.832$

$0.832 \times 8 = 6.656$

$0.656 \times 8 = 5.248$

$0.248 \times 8 = 1.984$

$0.984 \times 8 = 7.872$

$0.872 \times 8 = 6.976$

$\therefore (0.513)_{10} = (0.4065176)_8$

$\therefore (153.513)_{10} = (231.4065176)_8$

(c) Given that $(292)_{10} = (1204)_b$ determine 'b'

Given that $(292)_{10} = (1204)_b$

$\Rightarrow 292 = 1 \times b^3 + 2 \times b^2 + 0 \times b^1 + 4 \times b^0$

$= b^3 + 2b^2 + 0 + 4$

$= b^3 + 2b^2 + 4$

$\Rightarrow b^3 + 2b^2 = 292 - 4$

$\Rightarrow b^3 + 2b^2 = 288$

$\Rightarrow b^2(b + 2) = 288$

By trial and error method,

$b^2(b + 2) = 36 \times 8$

$b^2(b + 2) = 6^2(6 + 2)$

$\Rightarrow \boxed{b = 6}$

$\therefore (292)_{10} = (1204)_6$

Q18. Decimal system became popular because we have 10 fingers a rich person on earth has decided to distribute ₹ 1 lakh equally to the following persons from various planets. Find out the amount each one of them will get in their respective currencies:

A from planet VENUS possessing 8 fingers

B from planet MARS possessing 6 fingers

C from planet JUPITER possessing 14 fingers

D from planet MOON possessing 16 fingers.

Answer :

A person on earth has 10 fingers

$\Rightarrow$ Base = 10 on earth

$\therefore$ 1 lakh $= (100000)_{10}$

The person '*A*' is from Venus and has *B* fingers

$\Rightarrow$ Base = 8 on venus

Converting $(100000)_{10}$ to base 8.

8 | 100000
8 | 12500 – 0
8 | 1562 – 4
8 | 195 – 2
8 | 24 – 3
3 – 0 ↑

$\therefore (100000)_{10} = (303240)_8$

$\Rightarrow$ A gets the amount 303240

The person '*B*' is from Mars and has 6 fingers

$\Rightarrow$ Base = 6 on Mars.

Converting $(100000)_{10}$ to base 6,

6 | 100000
6 | 16666 – 4
6 | 2777 – 4
6 | 462 – 5
6 | 77 – 0
6 | 12 – 5
2 – 0 ↑

$\therefore (100000)_{10} = (2050544)_6$

$\Rightarrow$ *B* gets the amount 2050544

The person '*C*' is from Jupiter and has 14 fingers.

$\Rightarrow$ Base = 14 on Jupiter.

Converting $(100000)_{10}$ to base 14

14	100000
14	7142 – 12 or c
14	510 – 2
14	36 – 6
	2 – 8

$\therefore$ $(100000)_{10} = (2862C)_{14}$ where $C = 12$

$\Rightarrow$ C gets the amount $2862C$

The person 'D' is from Moon and has 16 fingers

$\Rightarrow$ Base= 16 on Moon

Converting $(100000)_{10}$ to base 16

16	100000
16	6250 – 0
16	390 – 10 or A
16	24 – 6
	1 – 8

$\therefore$ $(100000)_{10} = (186A0)_{16}$ where $A = 10$

Therefore, D get the amount $186A0$

3.1.2 Complements

Q19. What are the different types of complements? Explain.

Answer : Model Paper-I, Q6(b)

There are two different types of complements for any system with base 'r'. They are,

(i) (r – 1)'s complement

(ii) r's complement.

(i) (r – 1)'s Complement

$(r - 1)$'s complement of a number N in base 'r' and having n digits is,

$$\boxed{(r^n - 1) - N}$$

(ii) r's Complement

The r's complement of a number N in base 'r' and having n digits is,

$$\boxed{r^n - N}$$

The r's complement of N can also be obtained by adding '1' to its $(r - 1)$'s complement

i.e., $\boxed{[(r^n - 1) - N] + 1}$

For Decimal Numbers (base 'r' = 10)

The two types of complements for decimal numbers are,

(i) 9's complement i.e., $(r - 1)$'s complement

(ii) 10's complement i.e., r's complement.

(i) 9's Complement

The 9's complement of a decimal number N is defined as,

$$\boxed{(10^n - 1) - N}$$

Where 10^n = 1 followed by n 0s

Therefore, $10^n - 1 = n$ number of 9's.

Example

$10^5 = (100000)_{10}$ and $10^5 - 1 = 100000 - 1 = (99999)_{10}$

Thus, 9's complement of a decimal number N is obtained by subtracting each of its digits from 9.

The 9's complement of 25783 is 99999 – 25783 = 74216.

(ii) 10's Complement

The 10's complement of a decimal number N is defined as,

$$\boxed{10^n - N}$$

Where, 10^n = 1 followed by n 0s

In other words, the 10's complement of N can be obtained by keeping all least significant zeroes unchanged, subtracting the first non-zero least significant digit from 10 and then subtracting all the remaining higher significant digits from 9.

Example

The 10's complement of 25783 is obtained by subtracting the first non-zero least significant digit, i.e., 3 from 10 and then subtracting the remaining higher order digits i.e., 8, 7, 5 and 2 from 9. Thus 10's complement of 25783 is 74217.

An alternative method of finding 10's complement of N is adding 1 to its 9's complement.

The 9's complement of 25783 is 74216. Thus, the 10's complement of 25783 is 74216 + 1 = 74217.

For Binary Numbers (Base 'r' = 2)

The two types of complements for binary numbers are,

(i) 1's complement i.e., $(r - 1)$'s complement

(ii) 2's complement i.e., r's complement.

(i) 1's Complement

The 1's complement of a binary number N is defined as,

$$\boxed{(2^n - 1) - N}$$

Where, 2^n = 1 followed by n number of 0's

Therefore, $2^n - 1 = n$ number of 1s.

Example

$$2^5 = (100000)_2 \text{ and } 2^5 - 1$$
$$= 100000 - 1$$
$$= (11111)_2$$

In other words, 1's complement of a binary number N is obtained by subtracting each of its digits from 1. This subtraction changes all 0s to 1s and all 1s to 0s. Thus, the 1's complement of a binary number 'N' can be obtained just by changing 1s to 0s and 0s to 1s.

Example

The 1's complement of $(110110110)_2$ is $(001001001)_2$.

(ii) 2's Complement

The 2's complement of a binary number is defined as,

$$\boxed{2^n - N}$$

Where, 2^n = 1 followed by n number of 0s.

In other words, the 2's complement of a binary number N is obtained by keeping all its least significant zeroes and the first '1' unchanged, and then changing all the remaining higher order digits from 0s to 1s and 1s to 0s.

Example

The 2's complement of 110110110 is obtained by keeping the lower order 0 and 1 unchanged and then changing the remaining bits from 0s to 1s and 1s to 0s. Thus, the 2's complement of 110110110 is 001001010.

An alternative method of finding 2's complement of N is by adding 1 to its 1's complement.

Example

The 1's complement of 110110110 is 001001001. Thus, the 2's complement of 110110110 is 001001001 + 1, which is 001001010.

Q20. Find (72532 – 03250) using 9's complement.

Answer :

$(72532)_{10} - (03250)_{10}$

Let $A = 72532$

$B = 03250$

Then $A - B = A$ + (9's complement of B)

9's complement of B = 9's complement of 03250

$= (10^n - 1) - 3250$

$= (100000 - 1) - 3250$

$= 99999 - 3250$

$= 96749$

A + 9's complement of B = 72532 + 96749

$= (1)69281$

Carry

Since, a carry is generated, add the carry to the least significant bit of the result.

$$\begin{array}{r} 69281 \\ +\quad\ 1 \\ \hline 69282 \end{array}$$

Q21. Find the subtraction of 3250 and 72532 using 10's complement.

Answer :

For $A - B$,

Minuend (A) = 3250

Subtrahend (B) = 72522

We know that,

$A - B = A + r$'s complement of B.

Where, r's complement of $B = r^n - B$

$\Rightarrow$ 3250 – 72522 = 3250 + 10's complement of 72522

Where, 10's complement of 72522 $= 10^5 - 72522$

$= 100000 - 72522$

$= 27478$

$\therefore$ 3250 – 72522 = 3250 + 27478

= 30728

Since, it doesn't have any carry.

The solution will be the 10's complement of 30278 i.e., 9's complement of 30278 + 1

$\Rightarrow$ 69271 + 1 = 69272

3.1.3 Fixed-point Representation

Q22. Explain in detail about fixed-point representation.

Answer : Model Paper-III, Q6(a)

Fixed-point Representation

Mathematical conventions strategies is a conventional method. The modern electronics which are in use today should include these conventions in order to satisfy the demands of all the professionals. Number system is one of the branches of mathematics which has been promptly undertaken by the computer scientists to resolve many specifications. Also, working with unsigned numbers (positive numbers) is simple but while dealing with signed numbers (negative numbers) the real problem arises. In order to interpret signed numbers in terms of binary or computer readable data three conventions were deviced, these are,

(a) Signed magnitude representation

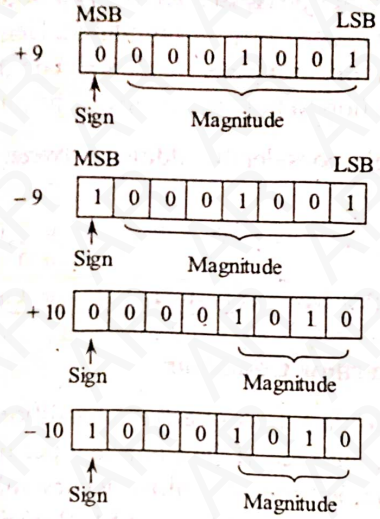
(b) Signed 1's complement representation and

(c) Signed 2's complement representation.

Detail analysis of these three representations is as follows,

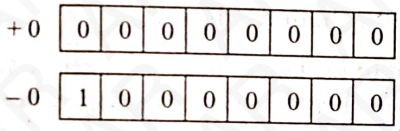
(a) Signed Magnitude Representation

This was the primary method of representation of signed numbers in terms of binary data. Here the convention used is that, if the sign of the given number is negative, or if a number is negative, then a digit '1' should be inserted at most significant bit position, followed by normal binary values. If the number is positive, or a number is carrying a positive sign, then, a digit '0' should be inserted at the most significant bit position, followed by the normal binary data. Consider the following representation of binary numbers 9 and 10 in terms of 8 bit registers.



Though the above representation looked fruitful, but it suffered two shortcoming with which, it is now obsolete. The shortcoming was that, whenever addition and subtraction operations were performed, signs of the numbers as well as their magnitudes were to be considered.

Secondly, there were two representation of a single digit zero (i.e.,)



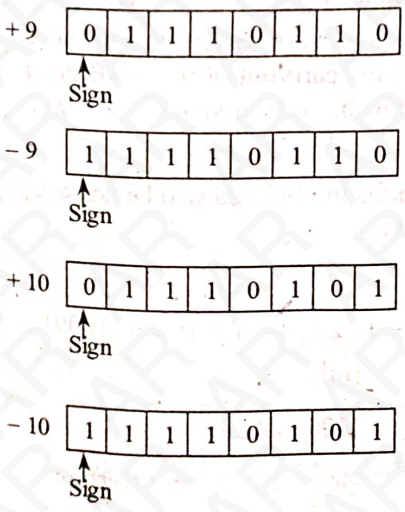
Which is not acceptable.

(b) Signed 1's Complement Representation

1's complement representation was brought up to replace the sign magnitude representation. But it did not serve longer. Many shortcomings overwhelmed even with this representation.

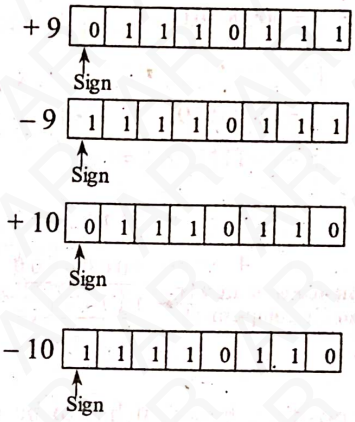
In order to represent a given signed number in terms of 1's complement representation, the magnitude of the number should be converted from 1 to 0 and 0 to 1. There will be no change in case of representation of sign of the number (i.e.,) in order to represent negative sign a value '1' is used at most significant bit position and a value '0' to represent the positive sign of the number.

For example consider the representation of decimal 9 and 10 in terms of 1's complement.



(c) Signed 2's Complement Representation

This is the most widely used representation when compared to above representations. The concept of 2's complement representation is simple which is an outcome of above two representations, i.e., sign magnitude and 1's complement. 2's complement is obtained by first computing 1's complement of the number and then adding 1 to the result. The representation of sign is same in all the three representations, i.e., a digit '0' to represent a positive number and '1' to represent a negative number. The magnitude of the number will be in 2's complemented form of the original binary data. Hence representation of 9 and 10 in 2's complement in terms of 8-bit register is given below.



Q23. Perform arithmetic on the following numbers using their binary equivalents.

(i) 5 + 4

(ii) – 5 + 4

(iii) 5 – 4

(iv) – 5 – 4.

Answer :

Consider the following arithmetic using 2's complement procedure. The rules to perform arithmetic is as follows.

Step 1

For any positive number, let it's representations be in a normal signed magnitude representation.

Step 2

In case of negative numbers, initially consider their 2's complement representation.

Step 3

Perform only addition (even though subtraction is prescribed to be performed).

Step 4

If the answer is negative, then the answer is in 2's complement form.

Step 5

Discard the digit, if there is a carry beyond the MSB (Most Significant Bit)

(i) 5 + 4

As both the numbers are positive, hence represent them in normal sign magnitude representation and perform addition.

$$\begin{aligned} +5 &\longrightarrow 00000101 \\ +4 &\longrightarrow 00000100 \\ +9 &\longrightarrow 00001001 \end{aligned}$$

(ii) −5 + 4

As '5' is negative, hence represent −5 in 2's complement representation and 4 in normal sign magnitude representation and perform addition.

$5 = 00000101$

$\boxed{-5 = \overline{5} + 1}$

$\overline{5} = 11111010$

$\overline{5} + 1 = 11111011 \Rightarrow -5$

$$\begin{aligned} -5 &\longrightarrow 11111011 \\ 4 &\longrightarrow 00000100 \\ \text{The required answer is in 2's complement} &\longrightarrow 11111111 \end{aligned}$$

(iii) 5 − 4

As '5' is positive, hence, it has to be represented in normal sign magnitude representation. As '4' is negative hence 2's complement representation should be used.

$4 = 00000100$

$\boxed{-4 = \overline{4} + 1}$

$\overline{4} = 11111011$

$\overline{4} + 1 = 11111100 \Rightarrow -4$

$$\begin{aligned} & \overset{1\ 1\ 1\ 1\ 1\ 1}{} \\ 5 &\longrightarrow 00000101 \\ -4 &\longrightarrow 11111100 \\ \text{The required answer} &\longrightarrow \textcircled{1}00000001 \\ \text{(i.e.,) binary equivalent of 1} & \end{aligned}$$

(iv) −5 − 4

As both the values are negative, hence consider the 2's complement of both 5 and 4 and then add.

$$\begin{aligned} & \overset{1\ 1\ 1}{} \\ -5 &\longrightarrow 11111011 \\ -4 &\longrightarrow 11111100 \\ \text{The required answer is in 2's complement} &\longrightarrow \textcircled{1}11110111 \end{aligned}$$

Q24. What is meant by overflow? Also explain how it can be detected.

Answer : Model Paper-I, Q7(a)

Overflow Condition

During certain addition operations, it may happen that the carry may occur beyond MSB (Most Significant Bit) position. Such situations are often referred to as overflow condition or situation. This concept should be considered, since, every computer possesses fixed sized registers and any carry beyond this cannot be accommodated. Hence in order to satisfy this condition, computers uses special circuitry which is referred as flip-flop to hold the overflow bit.

For example, consider the addition between −70 and −80.

$$\begin{aligned} & \overset{1\ 1}{} \\ -70 &\ 1\ 0\ 1\ 1\ 1\ 0\ 1\ 0 \\ -80 &\ 1\ 0\ 1\ 1\ 0\ 0\ 0\ 0 \\ \text{Overflow bit} \longrightarrow \textcircled{1} &\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 0 \end{aligned}$$

Detection of Overflow Condition

In order to detect an overflow condition, initially the addition has to be analyzed. If the addition is performed between two unsigned number, then overflow occurs at MSB position (i.e.,) a bit will be carried out beyond MSB position.

Overflow conditions can also be detected when addition takes place between two signed number. In these situations, two cases should be observed (i.e.,) carry at MSB and carry beyond MSB. If both of these carries are equal, then it can be concluded that an overflow has occurred and if both of these bits (carry at MSB and carry beyond MSB) are not same, then it can be concluded that overflow has occurred. This is possible by exclusive-ORing both the digits.

When addition takes place between a signed and an unsigned number, then overflow does not occur as the result will be always less than the largest of those two numbers.

Q25. Give means to identify on whether or not has overflow has occurred in 2's complement addition or subtraction operations. Take on example for each possible situation and explain. Assume 4 bit registers. If overflow register E is also considered, check whether the result is correct or not.

Answer :

An arithmetic overflow is said to occur when an arithmetic operation produces a result containing an additional bit i.e., a carry that cannot be placed in the representable range.

1. When Two Unsigned Numbers are Added

An overflow is said to occur if the carry out of the MSB position is 1. If it is 0, then overflow does not occur. The overflow register E holds the carryout of the MSB position. Thus, if $E = 1$, then overflow has occurred and if $E = 0$, overflow has not occurred.

And the result obtained will be correct, if the overflow bit is considered.

Example

(a) $(9 + 8) = 17$ i.e., $1001 + 1000 = 10001$

$$\begin{aligned} & 1 \quad 1001 \\ E \quad & 1000 \quad \text{(Add)} \\ \boxed{1} \quad & 0001 \quad E = 1 \Rightarrow \text{Overflow} \end{aligned}$$

(b) $(2 + 6) = 8$ i.e. $0010 + 0110 = 1000$

	11 0010	
	E 0110	(Add)
	0 1000	$= 8$ $E = 0 \Rightarrow$ No overflow

2. When Two Unsigned Numbers are Subtracted

An overflow occurs and overflow register E will hold 1. But, the result obtained is correct if the overflow bit in register E is not considered.

Example

$(9 - 2) = 7$ i.e., $1001 - 0010$

2's complement of $0010 = 1110$

$\Rightarrow 1001 - 0010 = 1001 + 1110$

	1 1001	
	E 1110	(Add)
	1 0111	$= 7$

3. When Two Signed Numbers are Added

An arithmetic overflow will not occur if two numbers have dissimilar signs. Because their sum will have a smaller value when compared to them, and thus can be placed in the representable range.

Example

	$+2 \rightarrow 0\ 010$	
	$-5 \rightarrow 1\ 011$	(Add)
	$-3 \rightarrow 1\ 101$	

$\Rightarrow$ $(+2) + (-5) = (-3)$

$\therefore$ No overflow

An arithmetic overflow may occur if they have same signs.

Example

	0 1			1 0 1	
$+6 \rightarrow$	0 1 1 0		$-6 \rightarrow$	1 0 1 0	
$+5 \rightarrow$	0 1 0 1	(Add)	$-5 \rightarrow$	1 0 1 1	(Add)
E →	0 1 0 1 1	$= -5$ (Overflow)	E →	1 0 1 0 1	$= +5$ (Overflow)

Considering the above examples, the addition of two positive signed numbers produced a negative signed number. And, the addition of two negative signed numbers produced a positive number. In both the examples, if the carry out of the sign bit (or MSB) is considered as an actual sign-bit, then the 5-bit result obtain will be correct, an overflow register E stores the overflow bit. But, a 5-bit result cannot be placed in a 4-bit register and thus the overflow.

4. When Two Signed Numbers are Subtracted

An arithmetic overflow will not occur if two numbers have the same sign. As their difference will have a smaller value when compared to them, and thus can be placed in the representable range as follows,

Example

1.

			1 1 1	
$+6 \rightarrow$	0 1 1 0		0 1 1 0	
$+5 \rightarrow$	0 1 0 1	(Sub)	1 0 1 1	(Add)
		$+1 \rightarrow$	0 0 0 1	

$\Rightarrow$ $(+6) - (+5) = (+1)$

$\therefore$ No overflow

2.

			0 0 0	
$-6 \rightarrow$	1 0 1 0		1 0 1 0	
$-5 \rightarrow$	1 0 1 1	(Sub)	0 1 0 1	(Add)
		$-1 \rightarrow$	1 1 1 1	

$\Rightarrow$ $(-6) - (-5) = (-1)$

$\therefore$ No overflow

An arithmetic overflow may occur if they have dissimilar signs.

Example

			0 1	
$(+6) - (-5) : +6 \rightarrow$	0 1 1 0		0 1 1 0	
$-5 \rightarrow$	1 0 1 1	(Sub)	0 1 0 1	(Add)
E →	0 1 0 1 1	$= -5$		

The difference obtained when (-5) is subtracted from $(+6)$ is (-5) which is not correct. If the carry out of the sign bit is considered as an actual sign bit then the 5-bit result obtained will be the correct one.

i.e. $01011 = +11$

But, a 5-bit result cannot be placed in a 4-bit register and thus, an overflow has occurred.

Alternatively, to identify whether an arithmetic overflow has occurred while adding/subtracting two signed numbers, the carry at MSB position and carry out of MSB position are applied to a XOR gate. If this gate produces a 0 as its output, then an overflow has not occurred. And, if it produces 1, then an overflow is said to be occurred.

3.1.4 Floating-point Representation

Q26. Explain in detail about floating-point representation. Support your answers with examples wherever necessary.

Answer : Model Paper-III, Q6(b)

Floating-point representation is essential since many scientific calculations include complex arithmetic in terms of floating-point numbers. Hence, the way, these floating-point numbers are represented internally in a given computer is referred to as floating-point representation.

Basically, any floating-point number, irrespective of its base (i.e., base 10 or base 2) is divided into two segments, with one segment representing the fractional and other representing the exponent part.

Following are the details of floating-point representation in case of decimal and binary numbers.

Floating-point Representation of Decimal Number

As mentioned earlier, the floating-point number is basically divided into two segments, with one segment representing the mantissa and other representing the exponent. Hence it takes the form as,

$$M \times K^e$$

Where 'M' corresponds to mantissa, 'e' corresponds to exponent and 'K' corresponds to radix.

For example, consider two floating-point numbers i.e., + 122.28 and + 5089.34

The floating-point number, i.e., 122.28 can be written as $+ 0.12228 \times 10^{+3}$ ($M \times K^e$). Where + 0.12228 forms the fractional part or mantissa, the power '3' is exponent and '10' corresponds to radix. Similarly +5089.34 can be written as $+ 0.508934 \times 10^{+4}$ ($M \times K^e$), where 0.5089 corresponds to fractional or mantissa part, + 4 corresponds to the exponent and '10' is the radix. While dealing with floating-point representation of decimal numbers one has to remember that the computer accommodates only the mantissa and exponents part including their signs in register while radix is just assumed.

Floating-point Representation of Binary Numbers

In floating-point representation, binary numbers are represented same as decimal numbers except the fact that binary numbers has base 2, hence its radix is base 2 (instead base 10 of the decimal number). Consider representation of + 1010.01 and –1000.10.

Here + 1010.01 is represented as $+ (0.1010010)_2 \times 2^{+4}$. In case of binary numbers, computer reserve 8 bit register to represent the mantissa or fractional value and 6-bit register to represent exponent value. Hence in registers the value of + 1010.01 is stored as,

0	1	0	1	0	0	1	0

R_1

and

0	0	0	1	0	0

R_2

The value in R_2 is binary equivalent of '4' since the exponent of 1010.01 is '4'. Also, the first value in R_1 is '0' which represents that the stored value is positive.

Now consider – 1000.10, this can be represented as $-(0.100010)_2 \times 10^{+4}$. These values can be stored in registers as,

1	1	0	0	0	1	0	0

R_1

0	0	0	1	0	0

R_2

It can be noticed from above representation that R_1 stores a value 1 at its MSB, this is because the given binary value is negative. Even while representing binary values, the computer follows the standard notation, i.e., $M \times K^e$, here the radix 'K' is '2'.

While dealing with floating-point representation, there is a concept referred to as normalization. A given value, (irrespective of its base) is said to be normalized if and only if there is a non-zero digit at the MSB position of the given number. Hence number 2897 is normalized and 02897 is not normalized. Similarly a number 1011011 is normalized and 01011011 is not normalized.

Q27. Explain how floating point number are represented. Represent the number $(+46.5)_{10}$ as a floating-point binary number with 24 bits the normalized fraction mantissa has 16 bits and the exponent has 8 bits.

Answer :

Representation of Floating-point Number

For answer refer Unit-III, Q26.

Floating-point Representation of $(+46.5)_{10}$

Given decimal number is $(+46.5)_{10}$

Converting it to the binary form,

46.5

2	46	2	5
2	23 - 0	2	2 - 1
2	11 - 1		1 - 0
2	5 - 1		
2	2 - 1		
	1 - 0		

$\Rightarrow (46.5)_{10} = (101110.101)_2$

Floating-point representation is of the form $M \times K^e$

Where, M = Mantissa, K = Radix and e = Exponent

We know that,

$\Rightarrow$ 101110.101 is the binary number

$\Rightarrow 1.01110101 \times 2^5$

$\Rightarrow 1.01110101 \times 2^{101}$

To obtain the normalized short real form of the number $1.01110101 \times 2^{101}$, bias the exponent by adding $(127)_{10}$ or $(1111111)_2$ to exponent (101)

$\Rightarrow$ 101 + 1111111 = 10000100

$\Rightarrow$ e = 10000100 (8-bits)

Thus, normalized short real form is,

$1.01110101 \times 2^{10000100}$

Thus, Mantissa is 01110101

Exponent is 10000100

The 16-bit mantissa stored in register is,

0111010100000000

The 8-bit exponent stored in register is,

10000100

Q28. What is meant by normalization in floating-point representation? Why do we need it? What is bias? What normalization is used in IEEE 754 standard?

Answer :

Normalization

While dealing with floating-point representation, there is a concept referred to as normalization. A given value, (irrespective of its base) is said to be normalized if and only if there is a non-zero digit at the MSB position of the given number. Hence, number 2897 is normalized and 02897 is not normalized. Similarly, a number 1011011 is normalized and 01011011 is not normalized. In general, user normalize floating-point numbers, since they give maximum possible precision.

Bias

A value $2^{(\text{number of bits in exponent})-1} - 1$ is said to be bias. It is a fixed value which is used in obtaining true exponent value. For example a bias value of 127 can have exponent values ranging from –127 to +127 respectively.

Normalization in Case of IEEE 754 Standard

Normalization process in case of IEEE 754 standard is observed while dealing with floating-point addition and subtraction operations. It initiates by shifting the consecutive bits towards their left. This process is continued till we are left out with a non-zero bit at most significant bit location. During shifting process there exists certain possibility of exponent under flow which can be significantly dealt before terminating the current operation.

Q29. Represent 32.75 and 18.125 in single precision IEEE 754 representation.

Answer :

IEEE 754 standard supports following format for representation of floating-point numbers.

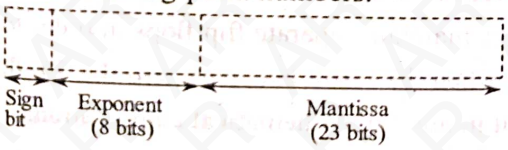


Figure: Single Precision IEEE 754 Standard Format for Representing Floating-point Numbers

(i) **32.75**

- As the above number is positive hence sign of the number is '0'.
- Initially, convert the above number into its binary equivalent value i.e.,

32	$0.75 \times 2 = 1.5$	0.5	1
16 – 0	$0.5 \times 2 = 1.0$	0.0	1
8 – 0			
4 – 0			
2 – 0			
1 – 0			
0 – 1			

Binary equivalent of 32 is = $(100000)_2$;

Binary equivalent of 0.75 is $(0.11)_2$.

Combine both the values.

$32.75 = (100000.11)_2$

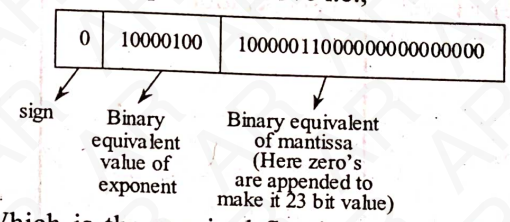
- Normalize above binary value,

  100000.11 (not normalized)

  1.0000011×2^5 (normalized value)
- The exponent value (i.e., 5) should be biased i.e., 127 should be added to it and it's equivalent binary value is calculated.

  $5 + 127 = 132$

  Binary equivalent of 132 is = 10000100
- Finally, represent all the values in the diagrammatic format specified above i.e.,



Which is the required floating-point value of 32.75 according to IEEE 754 standard.

(ii) **18.125**

- Binary equivalent of 18.125 is 10010.001
- Normalize the value,

  1.0010001×2^4
- Bias the exponent

  $4 + 127 = 131 = (10000011)_2$
- Represent in standard format

0	10000011	1001000100 0000000000 000

which is the required floating-point equivalent value of 18.125 as prescribed by IEEE 754 standard.

3.2 COMPUTER ARITHMETIC

3.2.1 Addition and Subtraction

Q30. How computer arithmetic addition and subtraction can be performed using signed magnitude data? Also provide it's hardware implementation.

Answer : **Model Paper-II, Q7**

The addition and subtraction procedures for unsigned numbers (positive numbers) is not a tedious process. Real complication arises while considering signed numbers (negative numbers). This is because, in case of signed numbers, there should be inclusion of additional circuitry which governs the changes in signs of the numbers before and after the operations performed. Also, the algorithm written should be sophisticated enough to tolerate these changes in signs and accordingly produce the result. When computer arithmetic is considered in case of signed-magnitude data eight different possibilities arise. These eight possibilities are illustrated below,

Sl. No.	Different Possibilities	Addition Operation	Subtraction Operation
1.	(+ve) + (+ve)	Both the numbers should be added and the result should be given a +ve sign.	No operation
2.	(+ve) + (–ve)	No operation	❖ If first number is greater than the second number then smaller number should be subtracted from the larger number the +ve sign should be given to the result. ❖ If first number is less than the second number then smaller number should be subtracted from the larger number and the result should be given a –ve sign.
3.	(–ve) + (+ve)	No operation	❖ If first number is greater than the second number then smaller number should be subtracted from the larger number and the result should be given a –ve sign. ❖ If first number is less than the second number then the smaller number should be subtracted from the larger and the result should be given a +ve sign.
4.	(–ve) + (–ve)	Both the numbers should be added and the result should be given a –ve sign.	No operation
5.	(+ve) – (+ve)	No operation	Same as condition 2.
6.	(+ve) – (–ve)	Same as condition 1	No operation
7.	(–ve) – (+ve)	Same as condition 4	No operation
8.	(–ve) – (–ve)	No operation	Same as condition 3.

While considering mentioned procedures, one has to remember that, if both the numbers are same and subtraction operation is performed, then the result will be +0. This is because, the computer always maintain separate flip-flops in order to store the signs of the numbers, while dealing with arithmetic of signed data. Hence, when the result becomes zero the flip flop store a positive value even though there is no significance of any sign appended to it in normal mathematical computations.

The following figure represents the hardware imple-mentation of computer arithmetic (addition/subtraction) using signed-magnitude numbers.

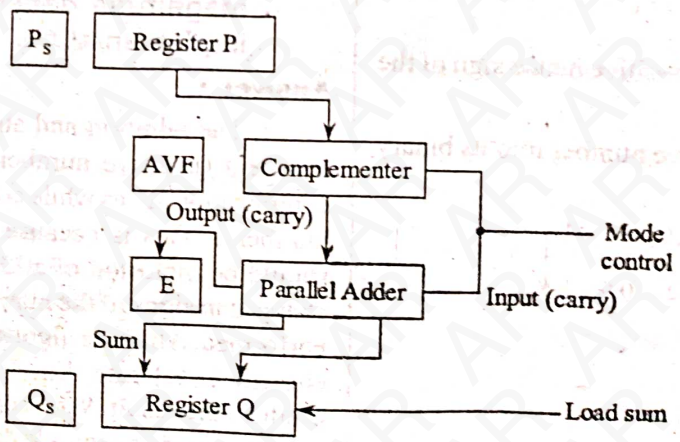


Figure: Circuitry (Hardware) Required for Signed-Magnitude Addition/Subtraction Operation

In the above representation, registers P and Q are used to store the two given numbers on which the required operations (addition/subtraction) are to be performed. Two sign flip flop P_{sign} and Q_{sign} with a label "sign" are used to store the signs of these numbers. Whenever, there is a requirement of addition operation to be performed on the given numbers, the parallel adder circuit is utilized and the result is stored in P register. Similarly, whenever subtraction operation is to be performed, the value of Q is 2's complemented and it is added to value of P and later the result is transmitted to register R which compares magnitude of two numbers. The add-overflow flip-flop E stores the value of carry which has occurred beyond MSB position. The parallel adder circuit maintains full adders internally and XOR gates form internal circuit of the complementer block. The operations of complementer depends on mode control line.

When the value of $M = 0$, the complementer transfers the value of B and input carry = 0 to the adder where the addition operation $A + B$ is performed. When the value of $M = 1$, then complementer transfers the complement of B and input carry = 1 to the adder where the addition operation $A + \overline{B} + 1$ is performed which is equivalent to subtraction operation $A - B$.

Q31. With the help of an example explain about addition and subtraction using signed 2's complement notation. Also provide its hardware implementation.

Answer :

Addition Operation using Signed 2's Complement Notation

Addition operation using signed 2's complement notation is same as normal addition operation. Here, the two numbers (which are to be added) are considered and converted into their binary equivalents. Later using the concepts of binary addition the given operation is performed. This analogy can be summarized as follows.

Step 1

Consider two numbers p and q.

Step 2

Obtain the binary equivalents of these two numbers. Here a sign bit should be included at MSB position of the binary representation. The value 0 at MSB position indicates that the given number is positive and a value 1 indicates the given number is negative.

Step 3

The binary representations of p and q are added by considering the rules of binary arithmetic.

Step 4

The bit which is exceeding the MSB position is discarded.

Step 5

End

Rules for performing binary addition is tabulated below,

i	j	i + j	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table: Rules for Binary Addition

Example

Consider the addition of (+34) and (+35)

Binary equivalents of +34 and +35 along with their signs are represented below,

+34 → 0 0 1 0 0 0 1 0

+35 → 0 0 1 0 0 0 1 1

```
          1                 1 <-- Carry
+34 -->   0  0  1  0  0  0  1  0
+35 -->   0  0  1  0  0  0  1  1
          ----------------------
+69 -->   0  1  0  0  0  1  0  1
```

Subtraction Operation using Signed 2's Complement Notation

Subtraction operation using signed 2's complement is different from normal subtraction operation. To perform subtraction of two numbers we need to first take the 2's complement of subtrahend and add the result to the minuend.

This is summarized as follows,

Step 1

Consider two numbers (say) p and q where p is minuend and q is subtrahend.

Step 2

Obtain the binary equivalents of these numbers along with their signs i.e., if these numbers are represented in the form of 8-bits, then a digit 1 at MSB position denotes that the given number is negative and a value 0 denotes that the given number is positive.

Step 3

Take the 2's complement of the subtrahend and add it to the minuend.

Step 4

If the step 3 results in an additional bit beyond MSB then the result is considered to be positive and the extra bit is discarded.

If the result of step 3 does not have any extra bit beyond MSB, then the result is considered to be negative and the result obtained is again converted into its 2's complement notation which is the required answer.

Example

Subtract

35 – 34

Minuend Subtrahend

Binary equivalent of these numbers along with their signs are represented below,

+ 35 → | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |

+ 34 → | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |

Obtain 2's complement of subtrahend.

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

2's complement of 34 → 0 1 0 1 1 1 1 0

Adding 2's complemented value to minuend.

```
+ 35 →                1 1 1 1 1 1   ← Carry
                      0 0 1 0 0 0 1 1
2's complement of (+ 34) → 1 1 0 1 1 1 1 0
+ 1 → (1)             0 0 0 0 0 0 0 1
```

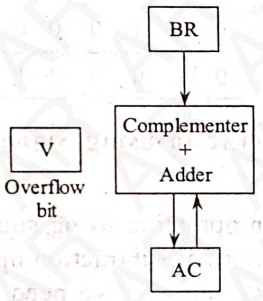


Figure: Hardware Representation of Signed 2's Complement Arithmetic (Addition & Subtraction)

In the above figure, two signed magnitude numbers are stored in the registers BR and AC. The left most bit in these registers signify the corresponding sign bits of the numbers. A complementer and parallel adder circuit is used in which addition or subtraction of the two sign bits with the other bits are performed. When an overflow occurs, the flip-flop *V* is set to 1, otherwise it is set to 0. However, the carry resulting from the output is neglected.

Q32. What is overflow and underflow? What is the reason? If the computer is considered as infinite system, do we still have these problems?

Answer :

Overflow

For answer refer Unit-III, Q24, Topic: Overflow Condition.

Underflow

An underflow occurs while performing subtraction operation i.e. whenever there is a borrow beyond most significant bit, we say that an underflow occurred. Whenever underflow is observed, it means that the result obtained is too small to be represented. Let us analyze this consequence by adhering to a simple example.

```
Indicates underflow →   1 1
                  A :   1 0 0 1
                  B :   1 0 1 0
              A – B :   0 1 1 1
```

Obviously if the computer is considered as infinite system there will not be any problems related to overflows since, the memory will be sufficient enough to manage overflow bits resulting out of computer arithmetic. But it is impractical to analyze such systems since the systems may get subjected to many unpredictable problems.

Q33. Explain how we can identify arithmetic overflow is occurred while adding/subtracting two signed numbers. Draw the circuit for performing addition/subtraction of two 4 bit numbers that checks the overflow.

Answer :

An arithmetic overflow occurs when an arithmetic operation produces a result containing an additional bit (i.e., a carry out of MSB position) that cannot be placed in the representable range.

The most significant bit of a signed number is regarded as its sign bit. Positive numbers have 0 as their sign bit whereas, negative numbers have 1 as their sign bit. The numbers are represented in 2's complement form.

Example

+ 4 has the binary code → 0 100

– 4 has the binary code → 1 100

+ 6 has the binary code → 0 110

– 6 has the binary code → 1 010

When Two Signed Numbers are Added

An arithmetic overflow will not occur if two numbers have dissimilar signs. Because their sum will have a smaller value when compared to them, and thus can be placed in the representable range.

Example

+2 → 0 010

–5 → 1 011 (Add)

–3 → 1 101

⇒ (+2) + (–5) = (–3)

∴ No overflow

An arithmetic overflow may occur if they have same signs.

Example

```
     0 1                          1 0 1
+ 6 → 0 1 1 0          – 6 → 1 0 1 0
+ 5 → 0 1 0 1 (Add)    – 5 → 1 0 1 1 (Add)
E → 0 1 0 1 1 = –5 (Overflow)   E → 1 0 1 0 1 = +5 (Overflow)
```

Considering the above examples, the addition of two positive signed numbers produced a negative signed number. And, the addition of two negative signed numbers produced a positive number. In both the examples, if the carry out of the sign bit (or MSB) is considered as an actual sign-bit, then the 5-bit result obtain will be correct, an overflow register E stores the overflow bit. But, a 5-bit result cannot be placed in a 4-bit register and thus the overflow.

When Two Signed Numbers are Subtracted

An arithmetic overflow will not occur if two numbers have the same sign. As their difference will have a smaller value when compared to them, and thus can be placed in the representable range as follows.

Example

1.
$$\begin{array}{llll} & & 1\ 1\ 1 & \\ +6 \rightarrow 0\ 1\ 1\ 0 & & 0\ 1\ 1\ 0 & \\ +5 \rightarrow 0\ 1\ 0\ 1 & (\text{Sub}) & 1\ 0\ 1\ 1 & (\text{Add}) \\ & & +1 \rightarrow\ 0\ 0\ 0\ 1 & \end{array}$$

$\Rightarrow$ (+6) – (+5) = (+1)
$\therefore$ No overflow

2.
$$\begin{array}{llll} & & 0\ 0\ 0 & \\ -6 \rightarrow 1\ 0\ 1\ 0 & & 1\ 0\ 1\ 0 & \\ -5 \rightarrow 1\ 0\ 1\ 1 & (\text{Sub}) & 0\ 1\ 0\ 1 & (\text{Add}) \\ & & -1 \rightarrow\ 1\ 1\ 1\ 1 & \end{array}$$

$\Rightarrow$ (–6) – (–5) = (–1)
$\therefore$ No overflow

An arithmetic overflow may occur if they have dissimilar signs.

Example

$$\begin{array}{llll} & & 0\ 1 & \\ (+6)-(-5): +6 \rightarrow\ 0\ 1\ 1\ 0 & & 0\ 1\ 1\ 0 & \\ -5 \rightarrow\ 1\ 0\ 1\ 1 & (\text{Sub}) & 0\ 1\ 0\ 1 & (\text{Add}) \\ & & \text{E} \rightarrow \boxed{0}\ 1\ 0\ 1\ 1 = -5 & \end{array}$$

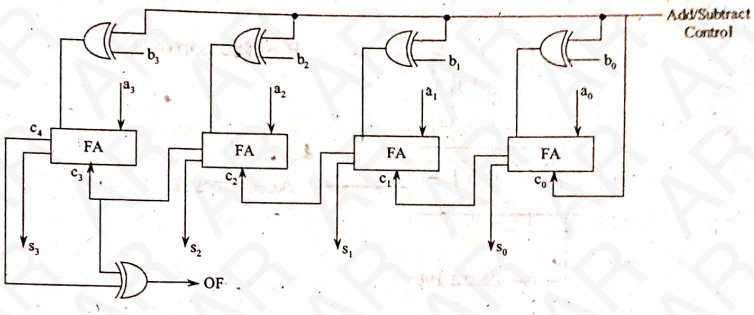
The difference obtained when (–5) is subtracted from (+6) is (–5) which is not correct. If the carry out of the sign bit is considered as an actual sign bit then the 5-bit result obtained will be the correct one.

i.e., 01011 = + 11

But, a 5-bit result cannot be placed in a 4-bit register and thus, an overflow has occurred.

Alternatively, to identify whether an arithmetic overflow has occurred while adding/subtracting two signed numbers, the carry at MSB position and carry out of MSB position are applied to a XOR gate. If this gate produces a 0 as its output, then an overflow has not occurred. And, if it produces 1, then an overflow is said to be occurred.

Adder/Subtractor Circuit that Checks for an Overflow



When add/subtract control = 0 addition is performed and when it is 1 subtraction will be performed. An overflow is said to occur when 'OF' is 1.

Q34. How many bits are needed to store the result addition, subtraction, multiplication and division of two n-bit unsigned numbers? Prove.

Answer :

In computing terminology, the best means for representing nonnegative number is binary format. Main advantage of such representation is the possibility of obtaining $2n$ different combinations of values for n bits. Using these combinations we can perform four different arithmetic operations i.e.,

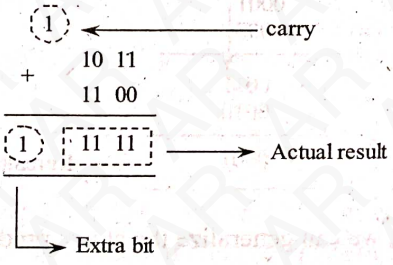
1. Addition
2. Subtraction
3. Multiplication
4. Division.

The requirement of number of bits for storing the results of mentioned operations varies with respect to the type of operation (i.e. addition, subtraction, multiplication, division etc). Let us generalize the above statement by considering few examples in this aspect.

1. Addition

For adding two n bit numbers we require $(n + 1)$ bits to store their result i.e. n bits for storing the actual result and an extra bit to store the possible carry (if any) generated.

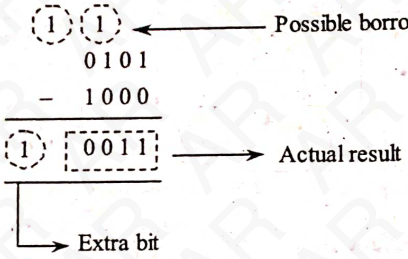
Example



2. Subtraction

Similar case is with the subtraction i.e. for performing subtraction between two n bit numbers, we require $n + 1$ bits for storing their result, i.e. n bits for storing actual result and an extra bit for storing the possible borrow.

Example



3. Multiplication

Here the scenario changes slightly i.e.,

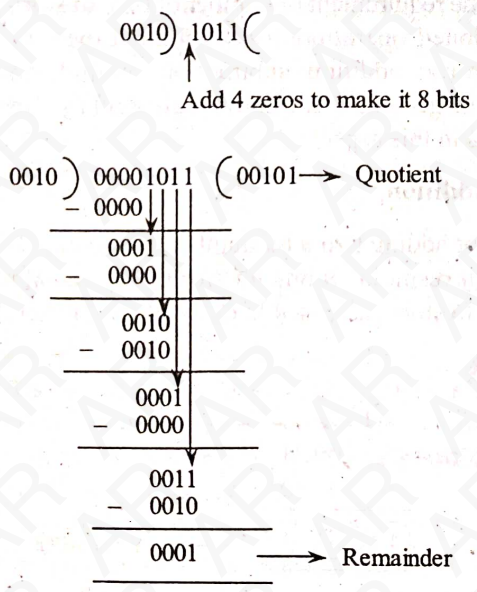
In order to perform multiplication between two n bit binary numbers, we require at most $2n$ bits for storing their result. This can be generalized by considering a small example.

$$\begin{array}{r} 1011 \\ \times\ 1110 \\ \hline 0000 \\ 1011- \\ 1011-- \\ 1011--- \\ \hline 10011010 \end{array}$$

4. Division

In order to perform division between two n bit binary numbers we require $(n + n)$ bits (n bits for storing the result (quotient) and n bits for storing the remainder).

Example



Now, we can generalize the above predictions as,

Addition	→ $(n + 1)$ bits
Subtraction	→ $(n + 1)$ bits
Multiplication	→ At most $2n$ bits
Division	→ $(n + n)$ bits.

Q35. Draw a flowchart to explain how addition and subtraction of two fixed point numbers can be done. Also, draw a circuit using full adders for the same.

Answer : Model Paper-I, Q7(b)

Flowchart representing the hardware implementation of adder/subtracter circuit is as shown in following figure,

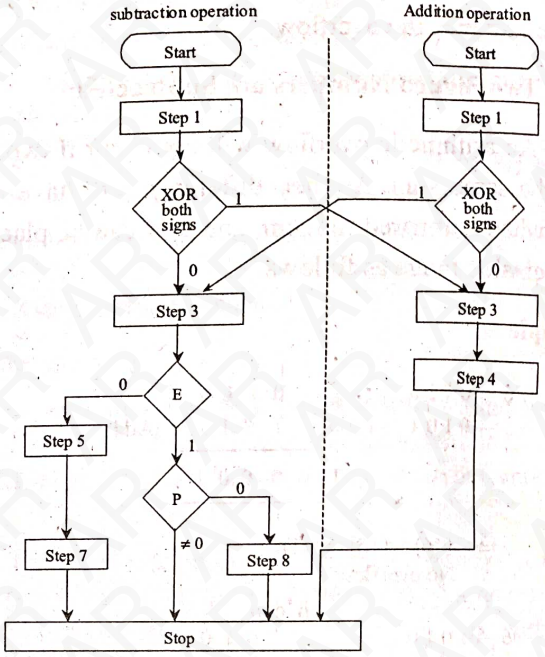


Figure: Flowchart for Signed-magnitude Addition/Subtraction

Following are the sequence of steps important to understand the above flowchart.

Addition Operation

Step 1

Let the two numbers be represented as P and Q.

Step 2

Here, the signs of these two numbers are compared by an XOR logic circuit. If the result is zero it is understood that the sign are same, else they are different. If two signs are same, the two numbers are added i.e. step 3 of addition operation, else they are subtracted i.e. step 3 of subtraction operation.

Step 3

Here, the values of P and Q are summed and the result is stored in RP register.

Step 4

The overflow bit which is stored in R is transferred to add overflow flip-flop E.

In short, the mentioned operations can be rewritten as,

Step 1

Define P, Q

Step 2

$P_{sign} \oplus Q_{sign}$

Step 3

$RP \leftarrow A + B$

Step 4

$E \leftarrow R$;

Subtraction Operation

Step 1

Let the two numbers be represented as P and Q.

Step 2

Here, the signs of these two numbers are compared by an XOR logic circuit. If the result is zero, it is understood that the signs are same, else they are different. If two signs are same, the two numbers are subtracted i.e. step 3 of subtraction operation else they are added i.e. step 3 of the addition operation.

Step 3

Here, the value of Q is subtracted with the value P (i.e. $P+\overline{Q}+1$) and the result is stored in a separate RP register which combines the values of R and P. Here the input is derived from two sources i.e. from step 2 of subtraction operation and step 2 of addition operation later the additional overflow flip flop is assigned a value zero.

Step 4

In this step the value of R is checked, this is to ascertain whether R is empty or contains the overflow value. If R is 0, it indicates that R is empty or if R is 1, it indicates that R contains an overflow value. Here if R is empty, step 5 is performed else step 6 is performed.

Step 5

When R is 0 it indicates that $P < Q$. Therefore, the value of P is complemented and is stored again in P.

Step 6 or C6

Here, the value in P is checked, if it is equal to zero, then step 8 is performed or if P does not contain a value zero then the process is directly terminated.

Step 7

In this step the value of P is added to excess 1 and the result is stored in P. Also the value of P_{sign} flip-flop is complemented and result is stored in the same flip-flop.

Step 8

A value zero is stored in the P_{sign} flip-flop and the process is halted.

In short, the above mentioned steps can be rewritten as,

Step 1

Define P; Q

Step 2

$P_{sign} \oplus Q_{sign}$

Step 3

$ER \leftarrow P + \overline{Q} + 1$

$E \leftarrow 0$

Step 4

Compare R

Step 5

$P \leftarrow \overline{P}$

Step 6

Compare P

Step 7

$P \leftarrow P+1$

$\text{sign}(P) \leftarrow \text{sign}(\overline{P})$

Step 8

$\text{sign}(P) \leftarrow 0$

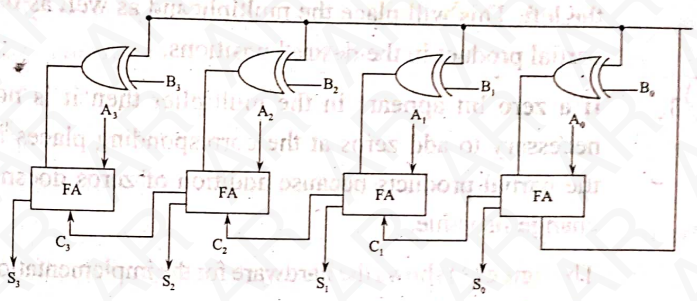


Figure: Hardware for Integer 4-bit Addition/Subtraction

3.2.2 Multiplication Algorithms

Q36. Draw a flowchart which explains multiplication of two signed magnitude fixed point numbers.

Answer :

The process of multiplying two signed-magnitude fixed-point numbers can be illustrated with an example which is given below,

Multiplicand 17 :					1	0	0	0	1
Multiplier 11 :					0	1	0	1	1
					1	0	0	0	1
				1	0	0	0	1	
			0	0	0	0	0		
		1	0	0	0	1			
	0	0	0	0	0				
Final product 187 :	0	1	0	1	1	1	0	1	1

The multiplication process begins by taking the least significant bit of the multiplier. If this bit is 0, then zeros are written at the corresponding places in the product and if it is 1, then all the bits in the multiplicand are written without any change. In successive lines, the numbers written are moved towards left by one position from the previous number. The process of multiplication continues till the most significant bit is encountered. Finally, all the numbers are added to generate the product of two binary numbers. By considering the signs of both multiplier and multiplicand the product's sign can be determined i.e., if both the signs are similar then the product's sign is positive, else it is negative.

Hardware Implementation

When the multiplication of binary numbers is done in a digital computer, its process changes in the following,

1. Instead of storing and adding the binary numbers in the registers, an adder is used to sum up the two binary numbers and a register is used for storing the partial products.
2. The partial products is transferred towards the right rather than shifting the numbers of the multiplicand to the left. This will place the multiplicand as well as the partial product in the desired positions.
3. If a zero bit appears in the multiplier then it is not necessary to add zeros at the corresponding places in the partial products because addition of zeros doesn't change its value.

The figure (1) shows the hardware for the implementation of multiplication.

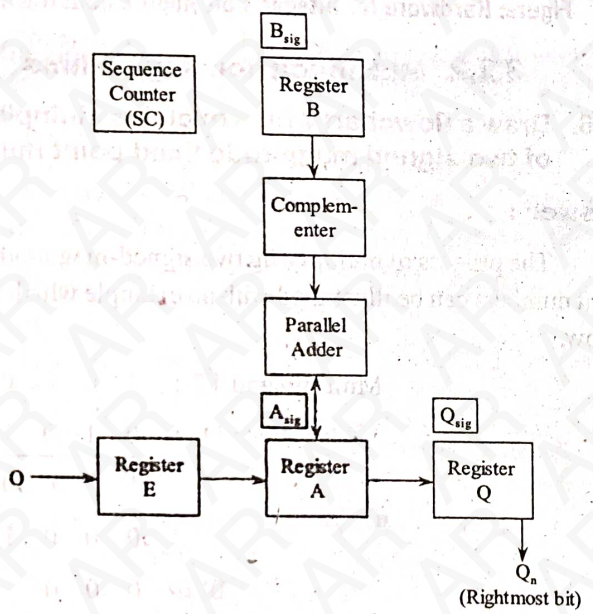


Figure (1): Hardware for Multiplication of Signed-magnitude Numbers

In the above figure, the multiplicand and multiplier are stored in the registers B and Q respectively. While the current partial products that is computed is stored in the register A. The three flip-flops A_{sig}, B_{sig} and Q_{sig} represent the corresponding signs of the numbers in the registers. The Sequence Counter (SC) holds a number which is equal to the number of multiplier bits in Q. Upon evaluating every partial products, the sequence counter is decremented by 1 and when this number becomes zero, the multiplication product is obtained and the process is stopped.

The multiplicand in B and the partial product in A are added and stored in the EA register. Then both partial product and multiplier undergoes a right-shift operation.

The right-shift operation is performed by shifting.

- A bit from E into the A's most significant position.
- A's least significant bit in the Q's most significant position.
- A zero in the register E.

Since, a bit of the current partial product is transferred to the Q register, the bits of the multiplier must be moved by one position towards the right thereby, storing the next bit (to be examined next) in a special flip-flop Q_n.

Flowchart

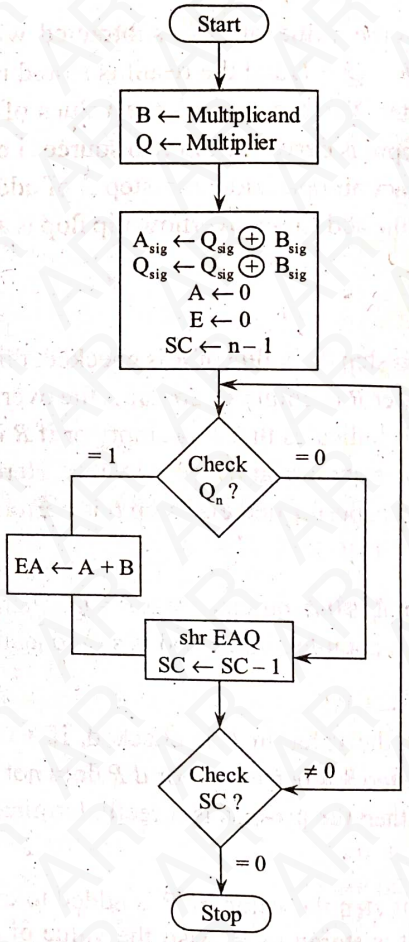


Figure (2): Multiplication of Two Signed-magnitude Numbers

The steps required for computing multiplication are as follows,

Step 1

Store the multiplicand in register B and multiplier in register Q.

Step 2

The signs of Q_{sig} and B_{sig} are XORed (i.e. they are compared) and the result is stored in flip-flops A_{sig} and Q_{sig}. Here, the register A and E are set to a value 0. The SC register has a number which is equal to the number of multiplier bits minus one (i.e., $n - 1$). Since, every operand has its corresponding sign which will occupy one bit.

Therefore, the magnitude will consist of $n-1$ bits.

Step 3

The value of least significant bit of multiplier in Q_n is checked. If $Q_n = 1$, then an addition operation is performed by taking the multiplicand (present in B) and current partial product (present in A), and the result is stored in EA register. If $Q_n = 0$ no operation is performed.

Step 4

A shift-right operation is performed on the register EAQ Q (i.e., shr EAQ) so as to generate a new partial product. After this operations SC is decremented by 1.

Step 5

The new value of SC is examined. If $SC \neq 0$ then the multiplication process is continued to generate new partial products. And, if SC = 0 the process is stopped and the final product is obtained with the result present in both A and Q registers.

Q37. Explain Booth's algorithm with its theoretical basis.

OR

What is a Booth's algorithm for 2's complement multiplications? Explain with an example.

Answer :

Model Paper-III, Q7

Booth's algorithms is the most strong and effective algorithm for solving signed 2's complement multiplication. It produces 2 n-bit product and gives equal priority to both positive and negative numbers. The algorithm adopts a techniques where in, it represents the multiplies as a difference between numbers. Thereby reducing the number of operations required for multiplication. It takes into consideration the fact that 0's in the multiplier require just shifting, and 1's in the multiplier from bit position n to bit position m can be treated as $2^{n+1} - 2^m$. For example, multiplier 00011110 (+30) can be represented as $2^5 - 2^1$ $32 - 2 = 30$. Therefore, the multiplication of multiplicand M and the multiplier 30 (i.e. $M \times 30$) is done as $M \times 2^5 - M \times 2^1$.

In Booth's algorithm, series of events can be described with the help of a flowchart as shown in figure (1).

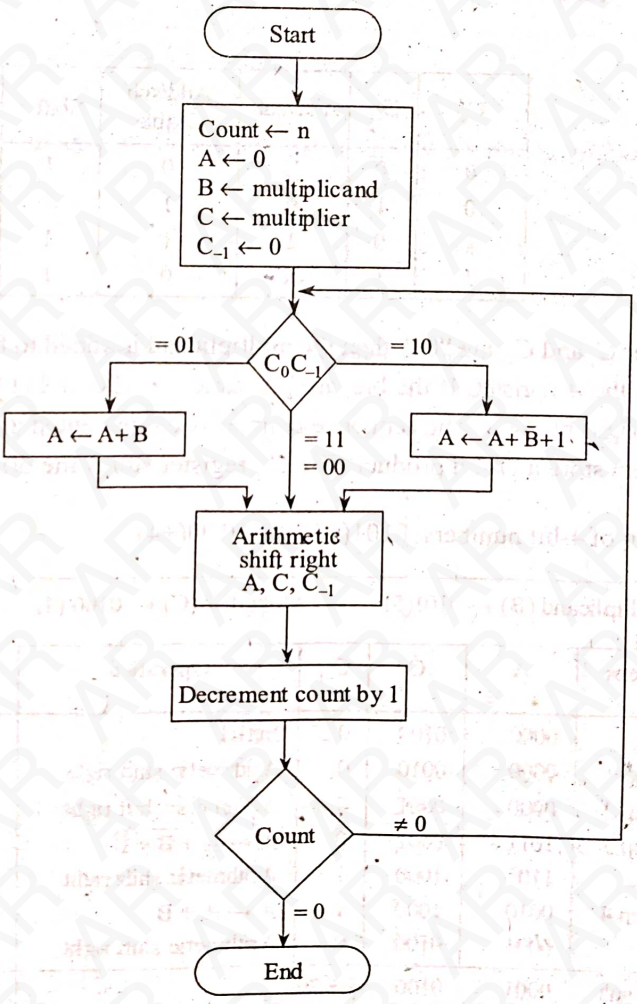


Figure (1): Booth's Algorithm for Signed Multiplication

Hardware Implementation

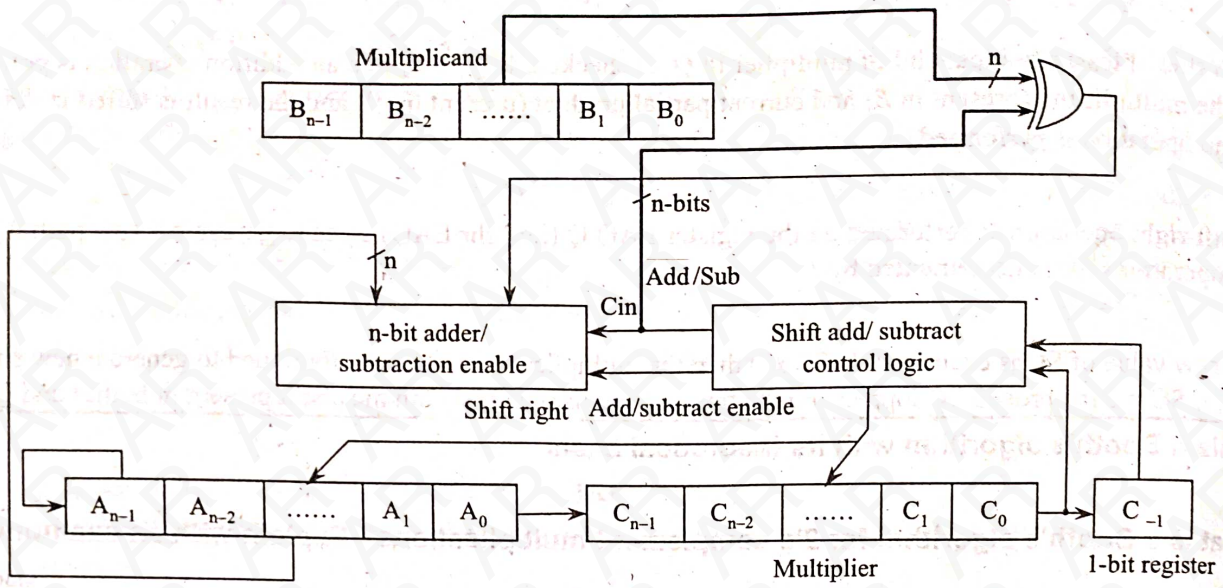


Figure (2)

The above figure represents the hardware implementation of booths algorithm. The circuit operater on *n*-bit adder, shift, add subtract control logic and four registers, *A*, *B*, *C* and C_{-1}. The C_{-1} is a 1-bit register that preserves the sign bit of a multiplier. At first the multiplicand and multiplier are loaded into registers *B* and *C* respectively, and then registers *A* and C_{-1} are initially set to 0. There is a sequence counter, count that is set to a number *n* equal to the number of bits in a multiplier. The role of the shift add/subtract control logic is to scans bits C_0 and C_{-1} one after the other and generates the control signals as shown in the following tables.

C_o	C_{-1}	Add/sub	Add/sub Enable	Shift
0	0	X	0	1
0	1	0	1	1
1	0	1	1	1
1	1	X	0	1

If the two bits of a multiplier C_0 and C_{-1} are "01" then the multiplicand is added to the *A* register and if they are "10" then the multiplicand is subtracted from the *A* register. If the bits are the same i.e., (00 or 11) then all of the bits of *A*, *C* and *D* and C_{-1} registers are made shift to the right by 1 bit. The sequence counter is decremented and the loop is repeated until the count becomes zero. The *A* and *C* registers store a 2*n*-bit product. The C_{-1} register stores the original sign bit of the multiplier.

Let us see the multiplication of 4-bit numbers, 0101(+5) and 0100(+4).

Multiplicand (B) ← 0101(5)		Multiplier (C) ← 0100 (4)			
Steps	A	C	C_{-1}	Operation	Count
	0000	0100	0	Initial	100
Step 1	0000	0010	0	Arithmetic shift right	011
Step 2	0000	0001	0	Arithmetic shift right	010
Step 3	1011	0001	0	$A \leftarrow A + \overline{B} + 1$	
	1101	1000	1	Arithmetic shift right	001
Step 4	0010	1000	1	$A \leftarrow A + B$	
	0001	0100	0	Arithmetic shift right	000
Result :	0001	0100	= +.20		

Q38. Represent two n-bit unsigned numbers multiplications with a series of n/2-bit multiplications.

Answer :

Multiplication of two n-bit unsigned numbers is performed by a series of $n/2$ bit multiplications, one n-bit addition and a single $2n$-bit addition. Let us assume, that P, Q are two unsigned n-bit numbers. Also assume that k, l are the Most Significant Bits (MSB) of P and Q and m, n are Least Significant Bits (LSB) of P and Q.

The multiplication of two unsigned n-bit numbers is given by,

$$PQ = (2^{n/2}k + m)(2^{n/2}l + n)$$

$$PQ = 2^n kl + 2^{n/2}(kn + ml) + mn \quad \ldots (1)$$

In equation (1), kl, kn, ml, mn represents a series of $n/2$ bit multiplications where each multiplication forms a n bit long number. $2^n kl$ and mn results in $2n$-bit number which is added to the summation of kn and ml with appropriate shifts by $2n$-bit adder.

Example

Take $n = 4, 0$

Let, $P = 0\ 1\ 0\ 1$

$Q = 0\ 0\ 0\ 1$

We have,

$$PQ = 2^n kl + 2^{\frac{n}{2}}(kn + ml) + mn = 2^4(00) + 2^2(01 + 10) + 11$$

Thus, two n bit (four bit) unsigned numbers multiplication are represented by series of $n/2$-bit (two bit) multiplications (00, 01, 10, 11).

Q39. Multiply 10111 with 10011 using Booth's algorithm.

OR

Multiply 10111 with 10011 using Booth's algorithm. Show all the registers content for each step.

Answer :

Multiplier (MR) = 10011

Multiplicand (MD) = 10111

2's complement of multiplicand = 01001

	AC	Multiplier (MR)	Q_{n+1}	$Q_n\ Q_{n+1}$	Sequential Counter (SC)
Initially	00000	10011	0		101
Subtract (MR)	01001			10	
	01001				
Shift Right	00100	11001	1		100
Shift Right	00010	01100	1	11	011
Add (MR)	10111			01	
	11001				
Shift Right	11100	10110	0		010
Shift Right	11110	01011	0	00	001
Subtract (MR)	01001			10	
	00111				
Shift Right	00011	10101	1		000

The multiplication result is stored in AC and MR register.

Q40. Show the step-by-step multiplication process using Booth algorithm when the following binary number are multiplied (– 7) × (+ 3).

Answer :

$+7 = 0111$
$-7 = 1000$ (1's)
$+ \quad 1$
1001 (2's complement)

$+3 = 0011$

Initialize,

$BR = 1001 \qquad QR = 0011$

$\overline{BR} + 1 = 0111, \quad Q_n = LSB(QR) = 1$

$Q_{n+1} = 0$

Q_n Q_{n+1}	Action	AC	QR Q_n	Q_{n+1}	Sequential counter
	Initially	0000	0011	0	4
1 0	AC ← AC + (BR + 1)	0000 +0111 0111	0011	0	
	Arithmetic Shift Right AC, QR, Q_{n+1}	0011	1001	1	3
1 1	Do nothing				
	Arithmetic shift right	0001	1100	1	2
0 1	AC ← AC + BR	0001 1001 1010	1100	1	
	Arithmetic shift right AC, QR, Q_{n+1}	1101	0110	0	1
0 0	Do nothing				
	Arithmetic shift right AC, QR, Q_{n+1}	1110	1011	0	0

Result = Contents of AC and QR

= 1110 1011

The result is negative, as indicated by its MSB (i.e., 1)

Therefore, user need to take 2's complement of the result.

Result = 11101011
1's complement = 00010100
$+ \quad 1$
2's complement = 00010101

$-(10101)_2 = (-21)_{10}$

Q41. What is the use of fast multiplication circuits? Write about array multipliers.

Answer :

Use of Fast Multiplication Circuits

Fast multiplication circuits are used when long multiplicand is given, (of size 1024 or 2048 bits or above). Performing bit-by-bit shifting operation on such a long multiplicand is computationally expensive.

Thus, to speed up the multiplication process fast multiplication circuits are used that provide fast adder structures in order to achieve optimal performance of n-bit by n-bit multiplication.

Array Multiplier

Array multiplier can be considered as one of the efficient layout of combinational circuits that efficiently perform the multiplication of two binary numbers. This multiplication can be done with one microoperation using a combinational circuit.

Working of Array Multiplier

Consider multiplicand and multiplier to be $M = m_1 m_0$ and $Q = q_1 q_0$ respectively.

The following are the steps for carrying out the multiplication of two 2-bit numbers.

1. The multiplicand bits $m_1 m_0$ are multiplied by the multiplier q_0 so as to generate PPO (first partial product).
2. The bit value of any part of partial product for instance $q_0 m_0 = 1$ only if $q_0 = 1$, $m_0 = 1$, otherwise it is equal to 0 (i.e., $q_0 m_0 = 0$). Since, this operation is similar to an AND operation, the multiplication can be implemented using an AND gate. Therefore, the first partial product is generated by using two AND gates.
3. The multiplicand bits m_1, m_0 are multiplied sequentially (one at a time) with the multiplier q_1 so as to generate PP1 (second partial product). During the multiplication process, the LSB (Least Significance Bit) of PP2 is shifted by one position towards left.
4. PP0, PP1 are added using two half-adder circuits. Instead of two half-adder circuits, it is preferable to use full-adder circuits for generating the sum because partial product may consist of more number of bits. The LSB of partial products are not affected by subsequent actions and therefore, they can be passed directly to the final product.

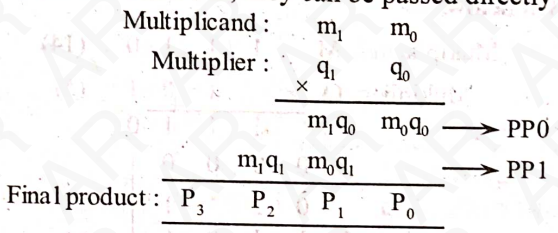


Figure (1): Multiplication of Two 2-bit Numbers

The figure (2) shows the implementation of an array multiplier with a combinational circuit for two 2-bit numbers.

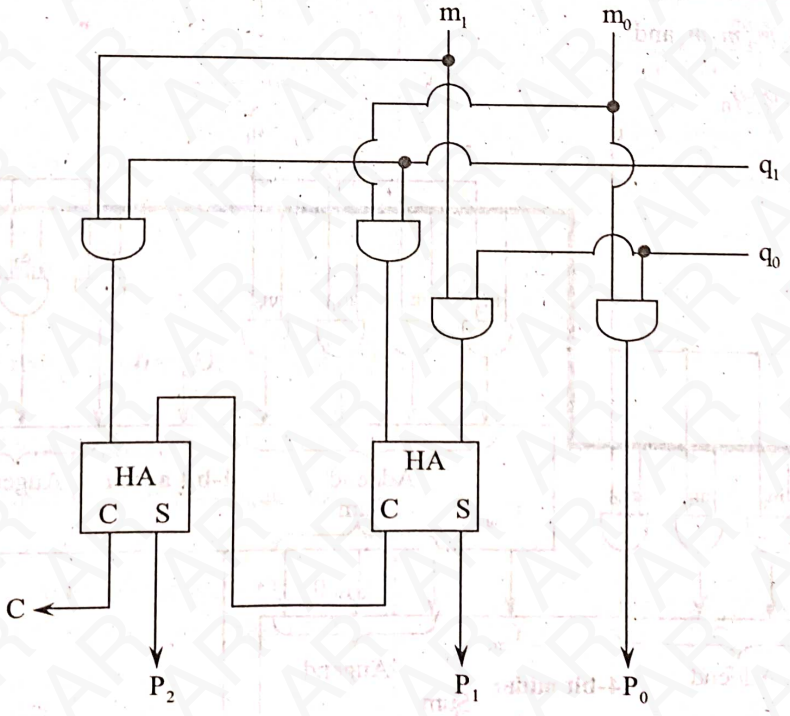


Figure (2): Array Multiplier for Two 2-bit Numbers

It is possible to design an array multiplier circuit for more number of bits in the similar fashion. Partial products can be generated by considering the bits of multiplier one at a time and adding them with each bit of multiplicand. The number of levels obtained during the multiplication depends on total number of multiplier bits. The output generated at each AND gate is added along with the partial product of the previous level in order to form a new partial product i.e. the first partial product PP0 can be generated using the AND of q_0 with each bit of M. The second partial product PP1 can be generated by performing AND operation on q_1 and M and adding it to PP0. Similarly, the same process can be continued for all the remaining partial products.

The last level gives the final product of the multiplication. As a general rule it can be said that if there are X multiplicand and Y multiplier bits then,

(i) The total number of AND gates required is equal to $(X * Y)$.

(ii) $(X-1)$ Y-bit adders are required so as to generated the final product of $X+Y$ bits.

Advantage

Array multiplier is one of the fastest means of carrying out multiplication of two binary numbers. The reason for this is that, the multiplier requires just the signal propagation time to form the multiplication array.

Disadvantage

Array multiplier was not considered as an economical approach because more number of gates were required. However, this problem was solved after the development of integrated circuits.

Q42. Draw a neat diagram for 4-bit by 3-bit array multiplier.

Answer :

Multiplication procedure is shown below,

```
Multiplicand: M      1  1  1  0   (14)
Multiplier:   Q  ×      1  0  1   (5)
                     1  1  1  0
                  + 0  0  0  0
                     0  1  1  1
               + 1  1  1  0
               1  0  0  0  1  1  0   (70)
```

Let us take, $M = m_3 m_2 m_1 m_0$ and

$Q = q_2 q_1 q_0$

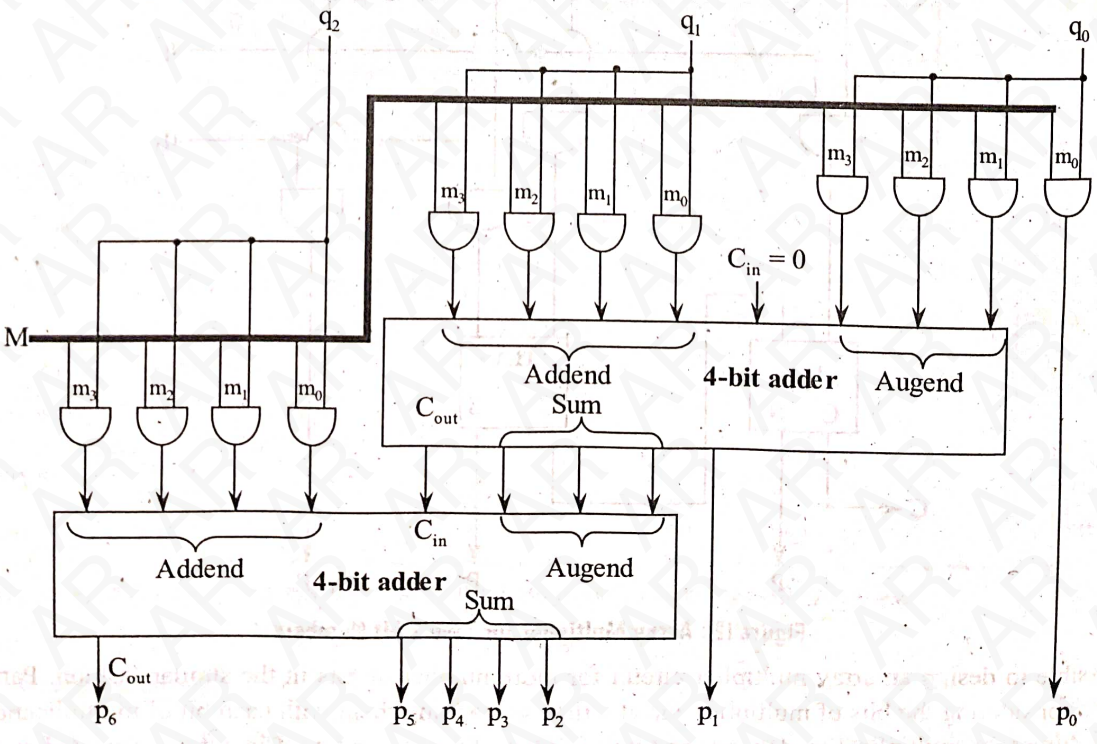


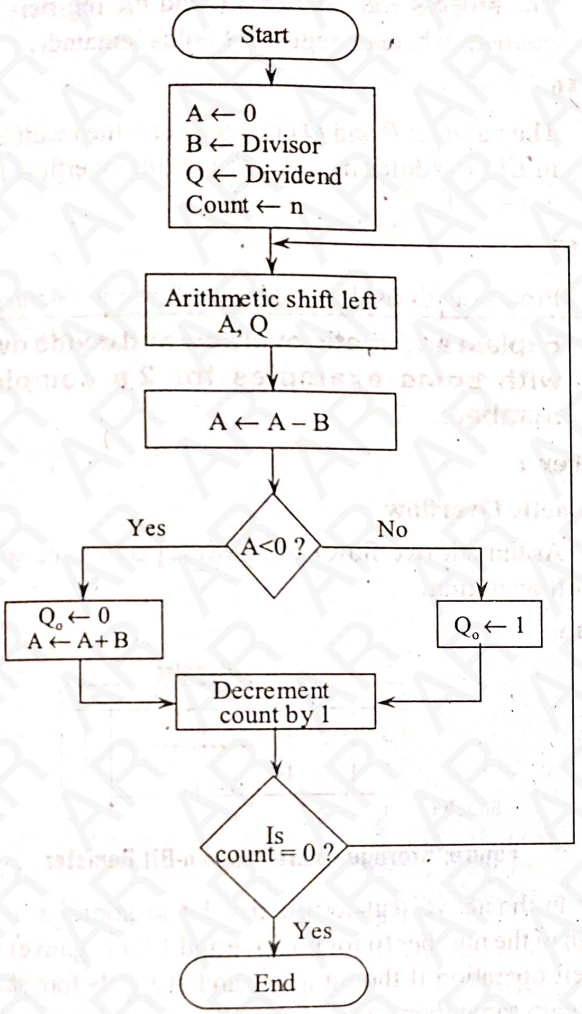
Figure: Array Multiplier for 4-bit by 3-bit

As shown in the above figure user require two 4-bit adders and 12 AND gates to implement the 4-bit by 3-bit array multiplier.

3.2.3 Division Algorithms

Q43. With the help of an example explain how binary division can be implemented using digital hardware.

Answer :



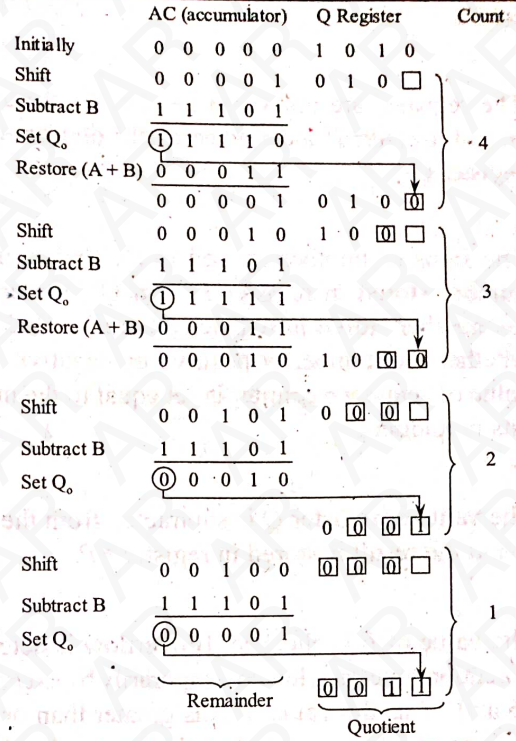
Figure

Division Operation Steps

1. Load the divisor into B register and dividend into A and Q registers.
2. Shift A and Q to its left binary position.
3. Subtract divisor from A and place answer back in A.
   $(A \leftarrow A - B)$
4. When the sign bit of $A = 1$ then, set the value $Q_o = 0$ and the divisor is added back to A or else the value of Q_0 is set to 1.
5. Repeat the above steps n times.
   At the end we get the quotient in Q and remainder in A.

Let us consider,

Q = Dividend
= 1010
B = Divisor
= 0011



Note: Subtract B means add B in 2's complement form.

Q44. With help of a flowchart explain the division operation for signed-magnitude data.

Answer :

Flowchart for a Division Operation

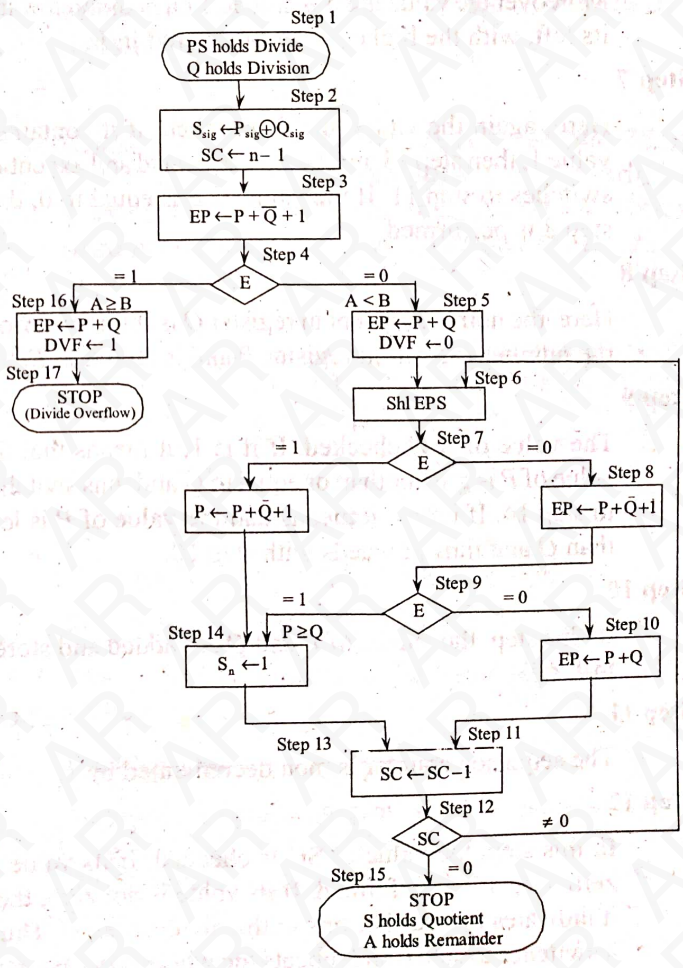


Figure: Flowchart for a Division Operation

Explanation

Step 1

The registers are initialized i.e., the dividend is stored in two registers P and S whereas, the divisor is stored in register Q.

Step 2

The signs of numbers stored in P_{sig} (belonging to the number stored in register P) and Q_{sig} (belonging to the number stored in register Q) are XORed to check whether the number is positive or negative. Also the value of sequence counter is set equal to the number of bits in quotient.

Step 3

The value of register Q is subtracted from the value of P and the result is stored in register EP.

Step 4

The value of E is checked. If overflow is detected, the execution process is halted temporarily by executing step 16 and 17 as the value of P is greater than or equal to Q. If no overflow is detected, that means P value is less than Q and the process proceeds smoothly by performing step 5.

Step 5, Step 6

The value of registers P and Q are added and the result is stored in EP. Later the divide overflow bit is set to 0. Moreover the value in PS is shifted 1 bit position towards its left, with the higher order bits stored in E.

Step 7

Here, again the value of E is checked, if it contains a value 1, then step 13 and 14 are performed and execution switches to step 11. If the value of E is equal to 0, then step 8 is performed.

Step 8

Here, the number present in register Q is subtracted from the number present in register P and is stored in EP.

Step 9

The value of E is checked. If it is 1, it means that the value of P is greater than or equal to Q and thus switches to step 14. If it is 0, it means that the value of P is less than Q and thus proceeds with step 10.

Step 10

In this step, the values of P and Q are added and stored in EP.

Step 11

The sequence counter is then decremented by 1.

Step 12

In this step, the value of SC is checked. If its value is zero, step 15 is performed. If its value is not zero, then it indicates that further bits in the dividend exist. Thus, it switches to step 6 and repeats the whole process again until the SC becomes zero.

Step 13

Q value is subtracted from P and is stored in P.

Step 14

The value in S_n is set to 1.

Step 15

The process ends as SC is 0 and the registers S holds quotient whereas, register A holds remainder.

Step 16

The value of P and Q are added and the result is stored in EP. In addition to that the divide-overflow flip-flop is set to 1.

Step 17

Process ends as the divide overflow has occurred.

Q45. Explain arithmetic overflow and divide overflow with some examples for 2's complement numbers.

Answer :

Arithmetic Overflow

Arithmetic overflow occurs while performing arithmetic shift left operation.

Example

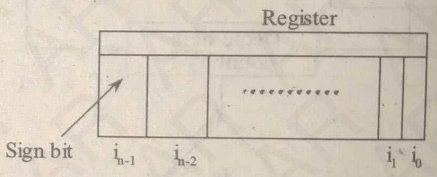


Figure: Storage Locations in n-Bit Register

In the above register the first bit location i.e., i_{n-1} holds the sign of the number (0 for positive and 1 for negative). During shift left operation if the bit at i_{n-1} and at i_{n-2} before shift does not remain same then such situation is referred to as arithmetic overflow. This is because after shift left operation the bit at i_{n-1} position will be claimed by the bit which was at i_{n-2} (before shift). This causes the bit at i_{n-1} (before shift) to be lost (after shift).

Example

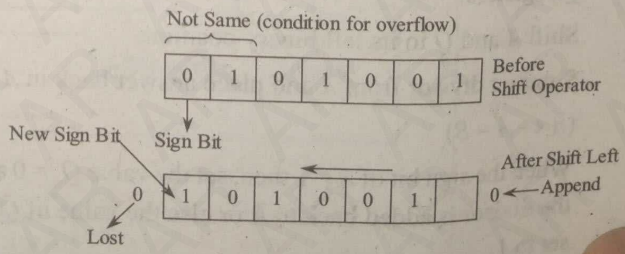


Figure: Arithmetic Overflow

Divide Overflow

While performing division operation, it sometimes happen that the register responsible for storing values corresponding to quotient falls short of space and hence the values get lost. In short, when the bits forming quotient exits the size of the register (prescribed for storing of bits) such condition is referred as "divide overflow".

While considering the condition in which the dividend is double the size of divisor, the above definition can be restated.

If the high-order half bits of the dividend forms a number which is greater than or equal to the devisor then such condition leads to divide overflow.

Example

Consider an example in which a system contains registers with 5-bits of length each. These registers are used to hold division, dividend and quotient. Since, dividend holds a larger value, two registers are used to store its bits. When the five most significant bits of the dividend constitute a number greater than or equal to the divisor then the number of bits in the quotient will be 6. But storing six bits in a standard 5-bit register is not possible. Hence, in this situation, the divide-overflow occurs.

Dividend : 1 1 1 1 0 0 0 0

1	1	1	1	0

0	0	0	0	0

Divisor : 1 0 0 0 1

1	0	0	0	1

Quotient : 1 1 1 0 0 0

1	1	1	0	0	0

↑ Divide-overflow bit

Remainder : 0 0 1 1 0

0	0	1	1	0

The problem of divide overflow must be prevented in case of normal computer operations since transferring more number of bits into a standard length of registers causes problems.

Q46. Perform the following divide operations,

(i) 110/111

(ii) 0011/1011.

Answer :

(i) 110/111

Dividend: 110

Divisor: 111

```
      0.110 (quotient)
111 ) 1100
      111
      ----
      1010
       111
      ----
       0110
       0000
      ----
        110 (Remainder)
      ----
```

$\therefore 110/111 = 0.\overline{110}$

(ii) 0011/1011

Dividend: 0011

Divisor: 1011

```
      0.010001011 (quotient)
1011 ) 001100
       1011
       ----
       000100000
          1011
          ----
          010100
          1011
          ----
          10010
          1011
          ----
          01110
          1011
          ----
          0111
          1011
          ----
          0011 (Remainder)
          ----
```

$\therefore 0011/1011 = 0.\overline{010001011}$

3.2.4 Floating-point Arithmetic Operations

Q47. What are the essential steps required to perform addition and subtraction operation on floating-point numbers? Explain in detail with help of a flowchart.

OR

Draw a flowchart to explain how two IEEE 754 floating-point numbers can be added, subtracted and multiplied. Assume single precision numbers. Give example for each.

Answer :

Addition or subtraction of two floating point numbers consists of the following four steps. The operands for addition/subtraction are stored in *AC* and *BR* registers.

1. Checking for zeros
2. Aligning the mantissas
3. Adding or subtracting two mantissas
4. Normalizing the output.

The algorithm computes the sum/difference in *AC* register.

1. Checking for Zeros

Whenever the inputs are provided, the algorithm initially verifies whether any one of the two given input values are equal to zero. If the value is zero, then it is not possible to normalize it and therefore the operation is terminated. It is preferable to carry out this step at the beginning instead of performing it in the middle of the normalization process.

2. Aligning the Mantissas

This step must be performed before performing any operations (like addition or subtraction) on the mantissas. The output generated after performing the operation can be unnormalized.

3. Adding or Subtracting Two Mantissas

This step is similar to the algorithm used for performing addition/subtraction of two fixed-point binary numbers. The magnitude part perform addition/subtraction by considering the following two factors.

- ❖ The operations to be performed
- ❖ The signs of two mantissas.

4. Normalizing the Output

Once an operation is performed on the mantissas, the unnormalized result must be normalized before transferring it to the memory.

The following flowchart shows addition/subtraction of two floating-point binary numbers.

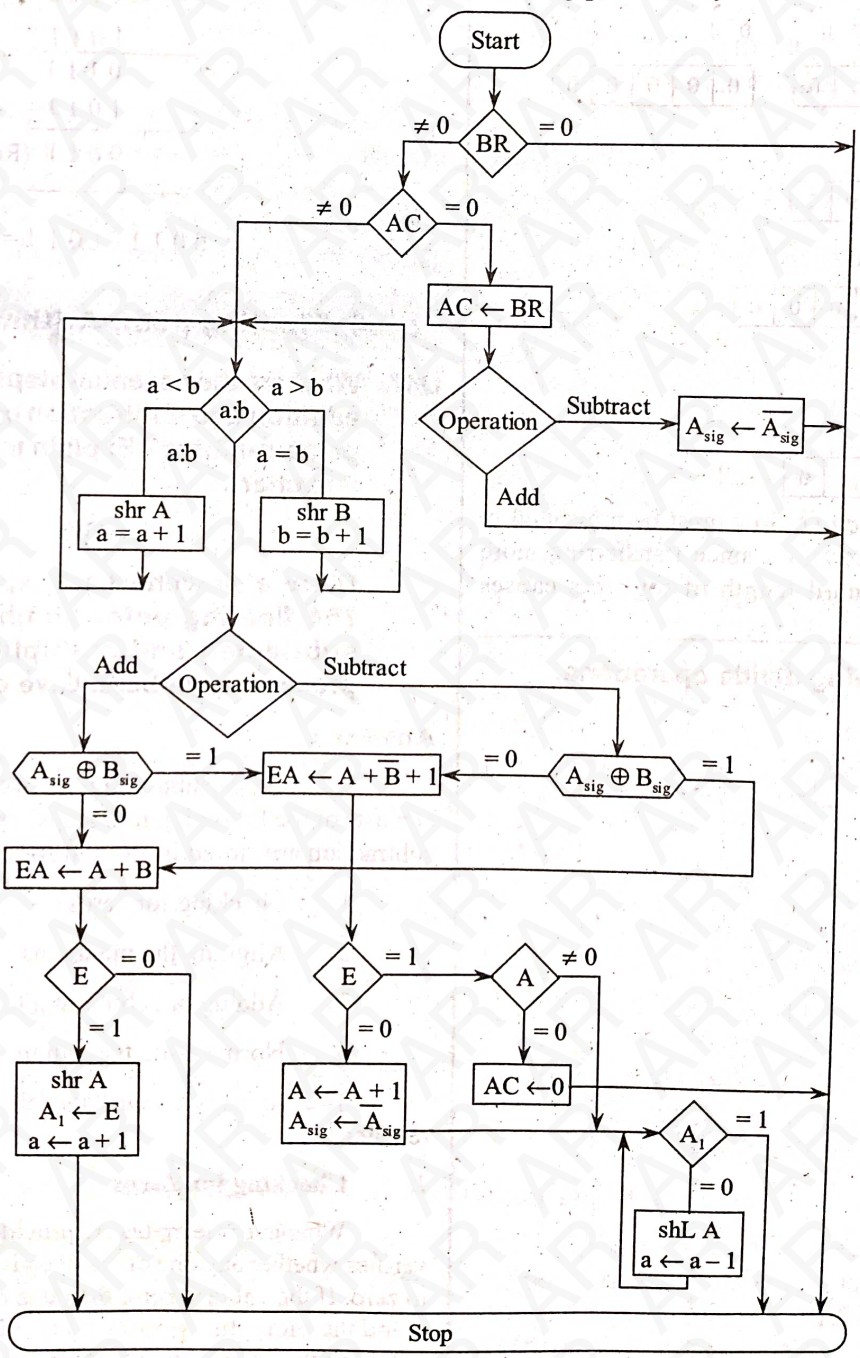


Figure: Flowchart for Floating-point Addition and Subtraction

Explanation

Step 1

Initially, the value present in register *BR* is checked. If the value is equal to zero, then the computation is terminated and the value present in *AC* is given as output.

Step 2

The value of *AC* register is verified. If it is equal to zero then the value stored in BR register is transferred to *AC* register. If subtraction is to be performed then the sign of *AC* register is complemented and the operation is terminated.

Step 3

If both floating-point numbers hold a nonzero value then the algorithm performs the alignment of mantissas.

Step 4

The magnitude comparator attached to exponents *a* and *b* generates three outputs. These outputs signify the relative magnitude of exponents. If $a = b$ then addition is carried out otherwise when $a \neq b$ the mantissa with a smaller exponent performs right-shift operation. Once the value is shifted to right, the exponent value of that mantissa is incriminated. This step is carried out one or more times and halts only when both the exponents become equal.

Step 5

The magnitudes of mantissas are added/subtracted depending on the operation to be performed and their sign bits are added and if an overflow condition is encountered then its value is transferred to the flip-flop *E*.

Step 6

The vale of *E* is verified. If $E = 1$ then the content of *E* is transferred in *A* and every bit present in register *A* is shifted to the right. On the other hand, if $E = 0$ then the operation is terminated. The value of the exponent is to be incriminated so as to generate an accurate number.

Step 7

During alignment process the mantissa that did not undergo shift operation is in the normalized form. Thus, it prevents the underflow condition from occurring.

Step 8

When the magnitude are subtracted then either of the following two conditions may occur.

(i) An Underflow Condition may be Encountered

The underflow condition occurs when the MSB of mantissa in $A_1 = 0$. In such situation, the mantissa undergoes left-shift operation and the exponent value is decremented. Then the value of A_1 is verified again to know whether it is equal to 1. If $A_1 \neq 1$ then the process is repeated until $A_1 = 1$. If the value of $A_1 = 0$, then the mantissa is normalized and the operation is terminated.

(ii) The Result Generated may be Equal to Zero

If the value of mantissa is equal to zero, then the number present in *AC* is made zero.

Q48. Write an algorithm to add binary number represented in normalized floating point mode.

Answer :

Algorithm to Add Floating Point Numbers

A common algorithm is used for performing both addition and subtraction of two binary numbers represented in normalized floating point mode. This is because both addition and subtraction are similar operations except for a sign change. Hence, the following algorithm can be used to perform addition of two normalized floating point numbers.

Algorithm

For answer refer Unit-III, Q47, Topic: Explanation.

Q49. Write an algorithm to multiply binary numbers represented in normalized floating point mode.

OR

How is floating point multiplication done?

Answer : **Model Paper-II, Q6(b)**

Multiplication Operation on Floating Point Numbers

In general, to perform multiplication or division on floating point numbers the computer rearranges the numbers in the form of,

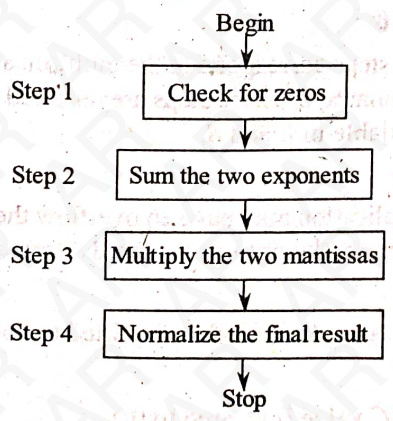
(Mantissa) × (radix)$^{\text{exponent}}$ or in short

$$M \times r^e$$

Hence, for example if one of the numbers to be considered for multiplication operation is 7.342. Then this number is rearranged by the computer in the form of 0.7342×10^1.

Where 7342 is the mantissa part, 10 is a radix and 1 is an exponent. Hence, in order to perform multiplication operation on these type of numbers, initially there mantissas are multiplied using normal rules of non-floating point numbers and later the two exponents are added and are placed above the radix.

Following are the basic steps required to perform floating point multiplication,



Following is a flowchart depicting the multiplication operation on floating point numbers.

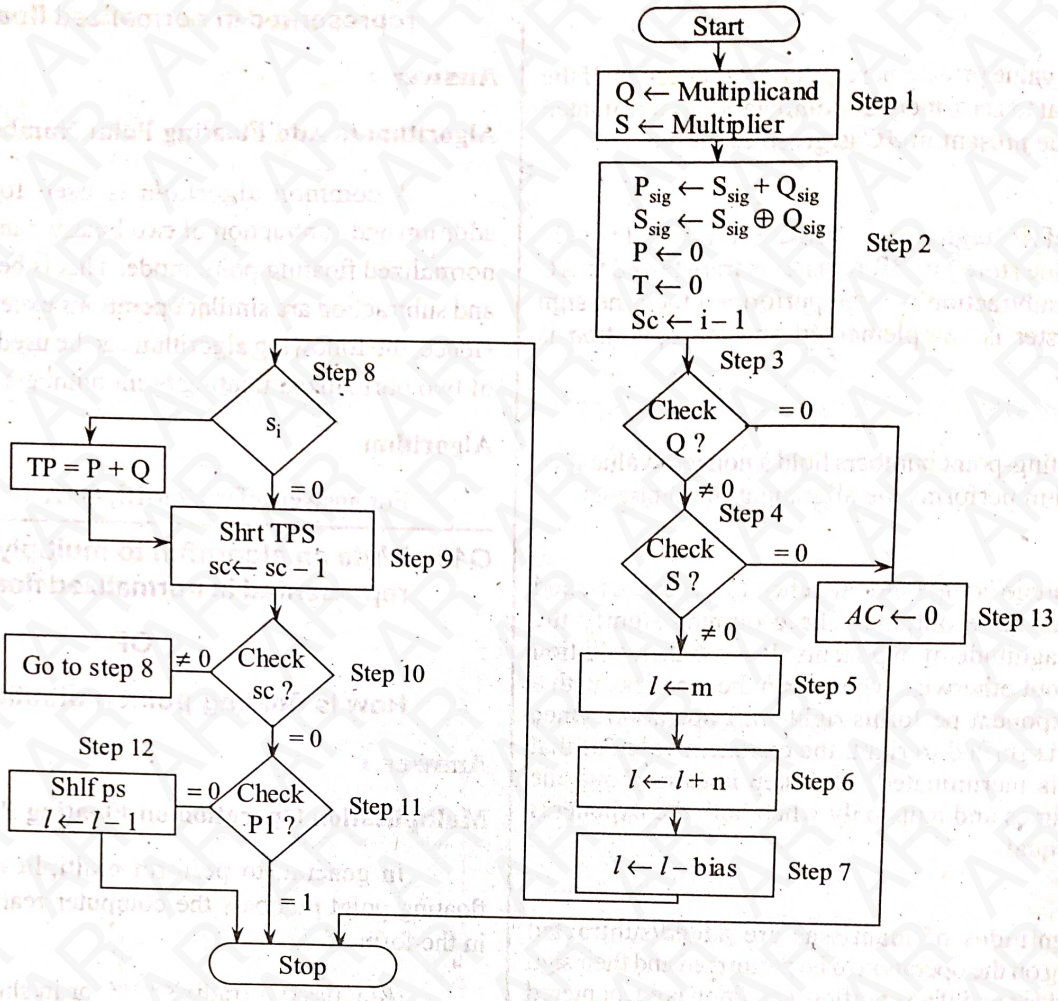


Figure: Multiplication Operation on Floating Point Numbers

Explanation

Step 1

The multiplicand is stored in register Q and the multiplier in S. Their respective signs gets stored in Q_{sig} and S_{sig} registers respectively.

Step 2

The S_{sig} and Q_{sig} are XORed to know whether the given number is (+ve) or (–ve). Several other initializations are also made.

Step 3, Step 4

The operands stored in Q, S are checked. If either of them is zero, the operation is halted by performing step 13.

Step 5, 6 & 7

In these steps the two exponents are added and the sum is placed in l. Later this sum is subtracted from bias in order to obtain the exact biased exponent.

Step 8, 9 & 10

These steps correspond to the multiplication of two mantissas. This multiplication is similar to the multiplication of fixed-point numbers. These steps are repeated until the value of sequence counter becomes (1). After the operation, the product is available in P and S.

Step 11

Multiplication may have an overflow therefore, the most significant bit in P is checked. If it is 0, the step 12 is performed. Otherwise, the product is already normalized and the final product is available in accumulator AC.

Step 12

In this step, the ps value is shifted to the left. That is, the l value is decremented by 1.

Step 13

The AC value is cleared to 0.

Q50. Explain how floating-point division is done.

Answer :

In general, to perform division or multiplication on any floating-point numbers, the computer rearranges these numbers in the form of,

(Mantissa) × (radix)$^{\text{exponent}}$ or in short

$$M \times r^e$$

Hence for example, if one of the numbers to be considered for division operation is 7.342, then this number is rearranged in form of 0.7342×10^1.

Where 7342 is the mantissa part, '10' acts as a radix and 1 is an exponent. Hence, in division operation the mantissas are divided and the exponents of two numbers are added to form the result. While dealing with division operation of floating-point numbers one has to remember that the procedure to perform floating point division is same to certain extent as compared to the procedure adopted to perform fixed point division. But the main difference arises at their storage requirements i.e., in the floating point division, the mantissa dividend is placed in a single fixed size accumulator register. As in case of floating point numbers, since the division has to be performed on the fractional parts (mantissa), hence the dividend mantissa must be placed in register p with the register s being empty, where the accumulation of zeros begins after the decimal point, thus these zeros has nothing to do during the time of execution.

Following flowchart represents the floating-point division.

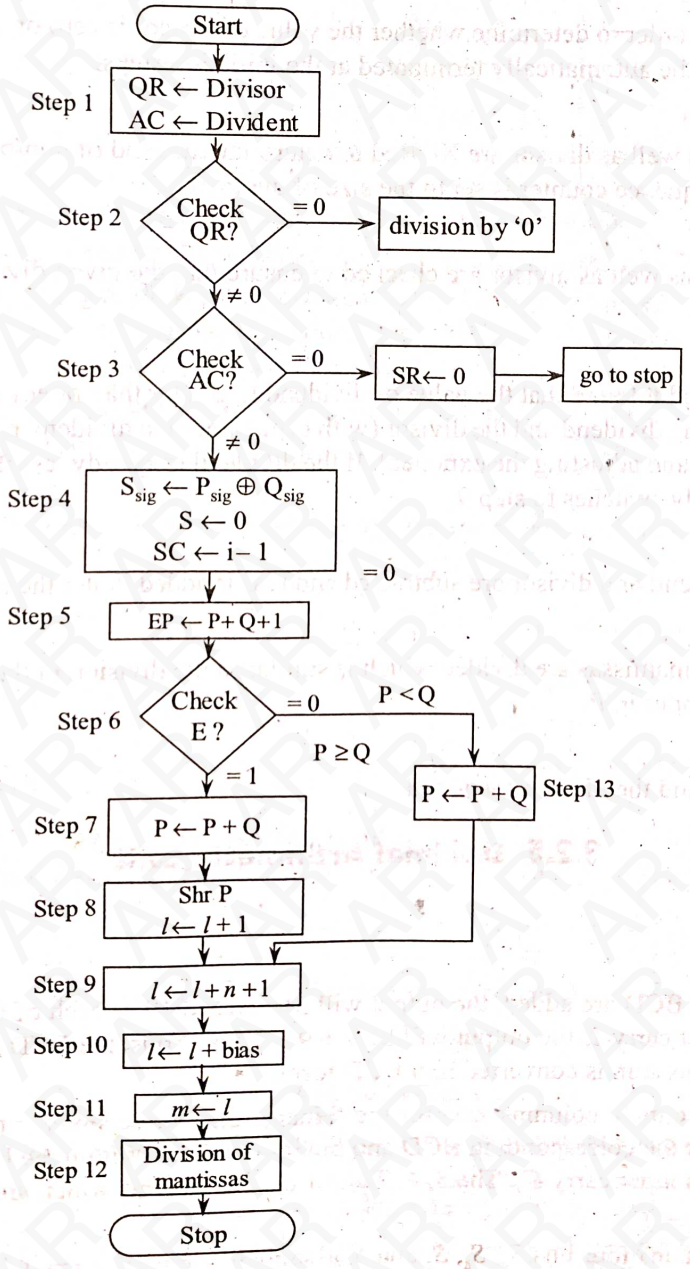
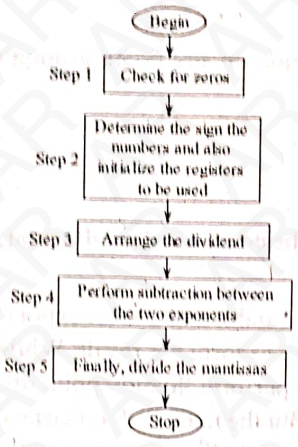


Figure: Floating-point Division

Various essential steps while performing floating point division are illustrated below,



Explanation

Step 1

The first step consists of initialization of various registers to be used.

Step 2 and Step 3

These steps are performed in order to determine whether the value of divisor is zero or, the contents of dividend are zero. If so, then the execution will be automatically terminated at the initiation steps.

Step 4

Here the signs of dividend as well as divisor are XORed to determine the kind of numbers (+ve or –ve). The register *s* is initialized to zero and the sequence counter is set to the size of quotient.

Step 5

Here the values of dividend as well as divisor are checked to ensure that the given divisor should be less than dividend. Its value gets stored in *E*.

Steps 6, 7 and 8

The value of *E* is checked and it found that the value of dividend is greater than or equal to the divisor, then the value of dividend is obtained by adding dividend and the divisor (with result stored in dividend register *p*) and shifting the contents of dividend towards its right and adjusting the exponent. If the dividend is already less than divisor, then after performing step 13, the execution directly switches to step 9.

Steps 9, 10 and 11

The exponents of both dividend and divisor are subtracted and bias is added. Later the result is transferred to *l*.

Step 12

Here, the magnitudes of the mantissas are divided which is similar to the division in the fixed-point case. The quotient is available in *S* and the remainder in *P*.

Step 13

The values of the dividend and the divisor are added.

3.2.5 Decimal Artihmetic Unit

Q51. Design a BCD adder.

Answer :

When two decimal digits in BCD are added, the output will not exceed 19 as each of the two bits can have at most 9 as their value. And, if there is an input carry 1, the output will be 9 + 9 + 1=19. Thus, the BCD adder will add the digits and will proceed the binary sum and then this sum is converted into BCD form.

In BCD derivation, there are three columns, one for the 'binary sum' the second for the 'BCD sum', the third column gives the decimal representation of the corresponding BCD and binary numbers. Within the binary sum column, there are four bits R_8, R_4, R_2 and R_1 along with an input carry C_2. The 8, 4, 2 and 1 are the weights which are assigned to each of the four bits that are present in a BCD code.

The BCD sum column contains four bits S_8, S_4, S_2 and S_1 along with an input carry C_1. When two decimal numbers are added, their output sum can be represented in BCD in the same form as the bits that appear in the BCD sum column.

Decimal value	BCD Sum					Binary Sum				
	C_1	S_8	S_4	S_2	S_1	C_2	R_8	R_4	R_2	R_1
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	1
2	0	0	0	1	0	0	0	0	1	0
3	0	0	0	1	1	0	0	0	1	1
4	0	0	1	0	0	0	0	1	0	0
5	0	0	1	0	1	0	0	1	0	1
6	0	0	1	1	0	0	0	1	1	0
7	0	0	1	1	1	0	0	1	1	1
8	0	1	0	0	0	0	1	0	0	0
9	0	1	0	0	1	0	1	0	0	1
10	1	0	0	0	0	0	1	0	1	0
11	1	0	0	0	1	0	1	0	1	1
12	1	0	0	1	0	0	1	1	0	0
13	1	0	0	1	1	0	1	1	0	1
14	1	0	1	0	0	0	1	1	1	0
15	1	0	1	0	1	0	1	1	1	1
16	1	0	1	1	0	1	0	0	0	0
17	1	0	1	1	1	1	0	0	0	1
18	1	1	0	0	0	1	0	0	1	0
19	1	1	0	0	1	1	0	0	1	1

As shown in the given derivation, the BCD and binary sums have the same values till the sum is of any value from 0000 to 1001. Thus, the conversion from binary to BCD is not required. When the binary sum exceeds 1001, it cannot be represented in a valid BCD form. To do so, 0110 (i.e., 6 in decimal) is added to the binary sum. This gives a correct BCD representation of the corresponding binary sum and produces an output carry 1.

The correction of the BCD sum is required when the R_8 bit is 1 and any of R_4 or R_2 bits is 1.

This can be expressed using the following boolean function,

$$C_1 = C_2 + R_8 R_4 + R_8 R_2$$

When C_1 value is 1, the BCD sum is corrected by adding 0110 to the binary sum and the output carry is forwarded to the next stage.

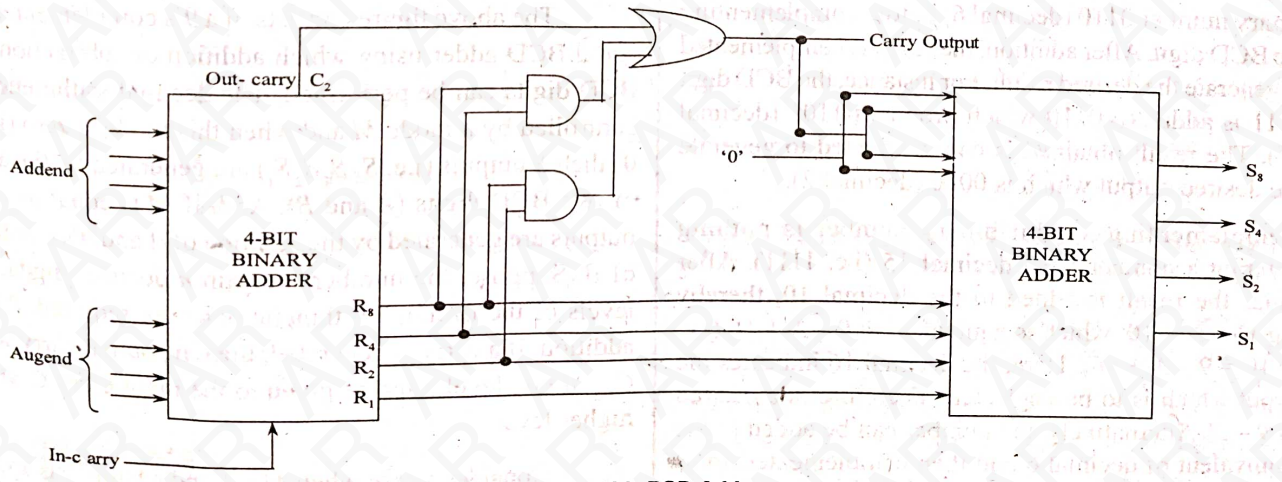


Figure: 4-bit BCD Adder

The above BCD adder adds the given BCD digits parallely and produces the sum as a BCD digit.

The correction logic is already included within the BCD adder. This is done, with the help of another 4-bit binary adder. The first 4-bit binary adder adds the given decimal digits by including the input carry as well. After adding, if the output carry remains 0, the correction is not required. But, when the output carry becomes 1, the second 4-bit binary adder adds the binary 0110 to the resulting binary sum to produce a valid BCD sum.

Q52. Explain the subtraction of decimal numbers in BCD.

Answer :

BCD subtraction can be done through the following circuits,

1. Subtraction circuit
2. Combinational circuit.

1. Subtraction Circuit

A subtraction circuit is used to find the subtraction of decimal numbers in BCD. It is more economical when it is performed by implementing the 9's or 10's complement of the subtrahend and adding the result to the minuend. However, the 9's complement of the decimal number cannot be obtained by simply complementing each bit in the number. This is because of BCD which is not considered as a self-complementing code. Therefore, the 9's complement of the decimal number is obtained by using a subtractor circuit that subtracts each BCD digit from 9.

To obtain the 9's complement of the decimal digits, the bits in BCD are complemented in the coded form only if a correction method is included.

The following are the two correction methods,

(i) In the first method, the 9's complement of the BCD digit is obtained by complementing each bit and adding to it the binary member 1010 (decimal 10). In this situation, if a carry occurs after the addition, then that carry needs to be neglected. For instance, the 9's complement of 0111 (decimal 7) is obtained by complementing it and adding the result 1000 to the binary number 1010 (decimal 10). After the addition, the carry is neglected and the result attained is 0010 (decimal 2).

(ii) In the second method, the BCD digit is added to the binary number 0110 (decimal 6) before complementing the BCD digit. After addition, the result is complemented to generate the desired result. For instance, the BCD digit 0111 is added to 0110 which produces 1101 (decimal 12). The result obtained is complemented to generate the desired output which is 0010 (decimal 2).

Complementing a 4-bit binary number is nothing but subtracting a number from decimal 15 (i.e. 1111). After subtracting, the result is added to the decimal 10, thereby producing $15 - N + 10$, which is equivalent to $9 - N + 16$ (i.e. $15 - N + 10 = 9 - N + 16$). Here, the decimal 16 indicates the carry output which is to be neglected. Therefore, the desired output is $9 - N$. Alternatively, the number can be added to the binary equivalent of decimal 6 and then complemented so as to produce the desired output i.e. $15 - (N + 6) = 9 - N$.

2. Combinational Circuit

A combinational circuit can also be used to find out the 9's complement of a BCD digit. When a BCD adder is combined with the combinational circuit, a BCD adder/subtracter is produced.

Consider an example in which B_8, B_4, B_2 and B_1 are the four binary variables which specify the subtrahend on addend and the variable M specifies a mode bit which is responsible for performing the add/subtract operation. If mode bit $M = 0$, then addition is performed and if $M = 1$ then subtraction is performed. The result obtained from the BCD 9's complementation circuit are denoted by the variables K_8, K_4, K_2 and K_1. The truth table of the circuit specifies that the variable B_1 must be complemented at all times and B_2 remains the same i.e., it does not change. It also specifies that after performing an exclusive-OR operation on B_2 and B_4, if the result obtained is 1, then $K_4 = 1$. Also, if $B_8 B_4 B_2$ is 000 then the variable $K_8 = 1$.

The BCD 9's complementer circuit has the following boolean functions.

$$K_1 = B_1 M' + B_1' M$$

$$K_2 = B_2$$

$$K_4 = B_4 M' + (B_4' B_2 + B_4 B_2')M$$

$$K_8 = B_8 M' + B_8' B_4' B_2' M$$

From these boolean expressions it can be observed that, if the mode bit $M = 0$, then $K = B$ and if $M = 1$, then different results of K are used to generate the 9's complement of B.

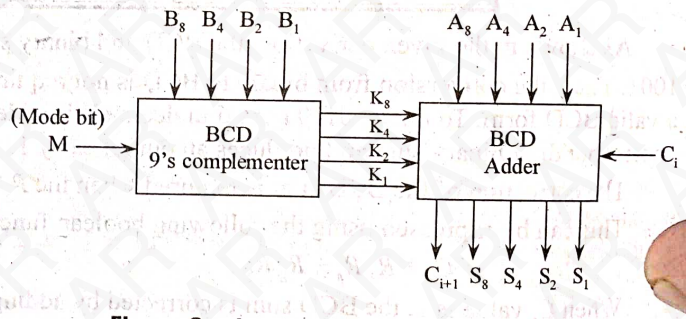


Figure: One Level of Decimal Arithmetic Unit

The above figure consists of a 9's complementer circuit and a BCD adder using which addition or subtraction of two BCD digits can be performed. The decimal arithmetic unit is controlled by a mode M and when this mode is zero (i.e. $M = 0$) then S outputs (i.e. $S_8 S_4 S_2 S_1$) are generated by the addition of two BCD digits (A and B). And, if M is equal to 1 then S outputs are generated by the addition of A and 9's complement of B. Suppose, the numbers contain n decimal digits, then n levels of the decimal arithmetic units are required. After the addition, if a carry is generated, then the output carry which is C_{i+1} at one level must be given to the input carry C_i at a next higher level.

Consider a case when $M = 1$ and a value 1 is applied to the input carry C_1 of the first level. This method is considered as an efficient method for subtracting two decimal numbers. In this case, S outputs are produced by adding A and 10's complement of B. The 10's complement of B performs a subtraction operation provided the carry of the final level is neglected.

3.2.6 Decimal Arithmetic Operations

Q53. With the help of diagrams explain the following,

(a) Parallel decimal addition

(b) Digital-serial, bit-parallel decimal addition

(c) All serial decimal addition.

Answer :

(a) Parallel Decimal Addition

The diagrammatic representation of parallel decimal adder is shown in figure (1).

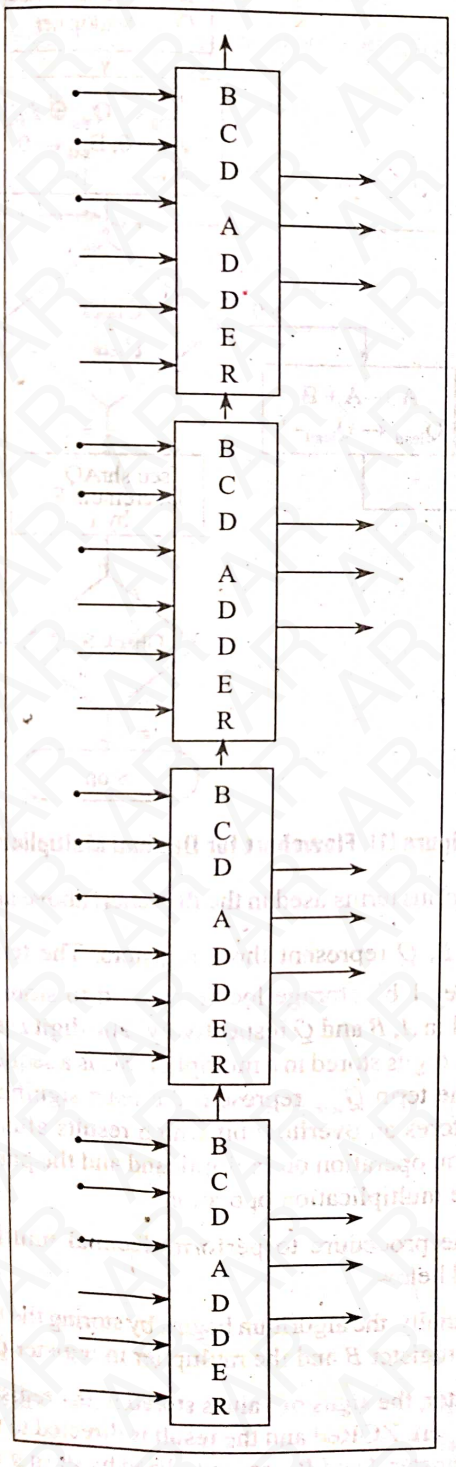


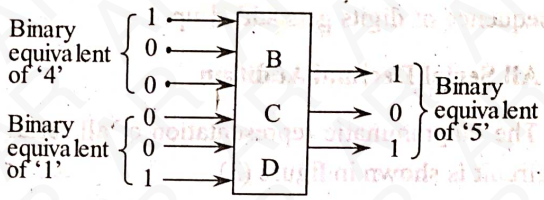
Figure (1): Representation of Parallel Decimal Adder

As shown in figure (1), the parallel decimal adder uses a decimal arithmetic unit that consists of a series of BCD adders, cooperating with each other by supplying required values and hence, generating the desired result. Here, while dealing with parallel decimal adders, one has to remember that, the succession of BCD adders increases with the increase in number of digits to be added. Hence for example, if the required number to be added is 3214 + 1231, then the circuit required is represented above.

The process of addition takes the following form.

3	2	1	4
1	2	3	1

As the digit 4 gets added to digit 1, hence the BCD adder for the corresponding addition can be represented as,



If the addition results in a carry, it is supplied to next successive BCD adder. This process goes on till the required result is obtained.

(b) Digital-Serial, Bit-Parallel Decimal Addition

The diagrammatic representation of digit-serial, bit-parallel decimal adder is shown in figure (2).

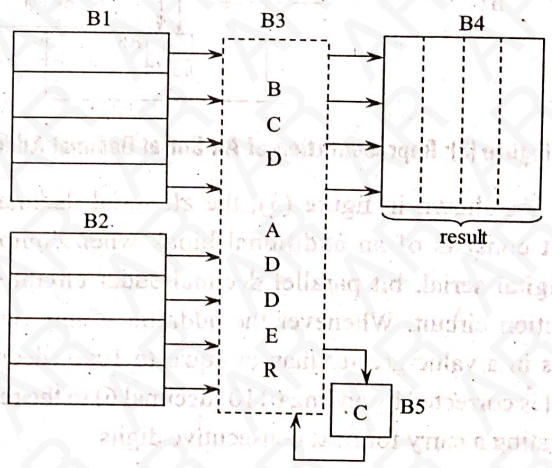


Figure (2): Representation of Digital Serial, Bit Parallel Decimal Adder Circuit

As shown in the figure (2), the digital-serial, bit-parallel decimal adder circuit consists of five major blocks, denoted by B1, B2, ... B5 respectively. The B1 and B2 are registers to store the augend and addend, B3, is a BCD adder, B4 is a register to store the result after addition and B5 is a 1-bit register to store the carry bit. As the name suggests, the circuit (BCD adder) accepts the data serially and produces the digits (after summing) parallely.

Consider an example where digits to be added be,

	3	2	1	4
+	1	2	3	1

Consider the first, addition operation i.e., the addition of 4 and 1. The binary equivalents of these digits gets stored in the B1 and B2 blocks respectively.

Later, these bits are applied to the BCD adder sequentially where the corresponding binary addition is performed. Finally the result of this operation is transmitted to B4 block. As the addition operation did not result in a carry bit generation, hence the value of carry remains zero. The process goes on till the entire sequence of digits gets added-up.

(c) All Serial Decimal Addition

The diagrammatic representation of all serial decimal adder circuit is shown in figure (3).

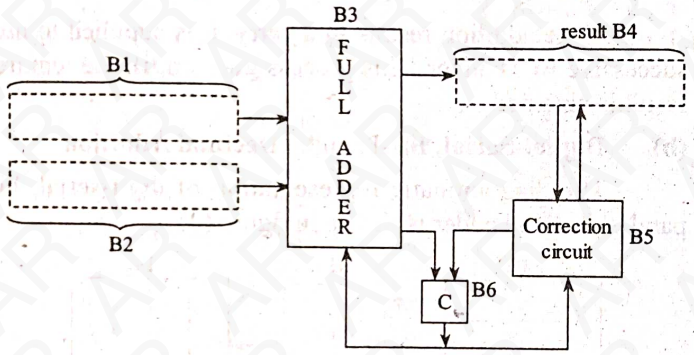


Figure (3): Representation of All Serial Decimal Adder

As shown in figure (3), the all serial decimal adder circuit consists of an additional block when compared to the digital serial, bit parallel decimal adder circuit i.e., the correction circuit. Whenever the addition of any two digits results in a value greater than or equal to 1010 (decimal 10) then it is corrected by adding 0110 (decimal 6) to the result and generating a carry for next consecutive digits.

Consider the addition of following numbers.

$$\begin{array}{r} 2\ 4\ 6\ \boxed{8} \\ +\ 1\ 2\ 3\ \boxed{4} \\ \hline \end{array}$$

When the digits 8 and 4 are added, the result obtained is 12 (1100) which is greater than 10 (1000). Hence, we add 0110 to it i.e. 1100 + 0110 = 10010. The result is stored in block B4 and the carry is transmitted to block B6 and for next consecutive digits. This process is performed by the correction circuit. Rest of the operation proceeds in the normal way i.e. each bit belonging to augend (B1) and addend (B2) circuit gets transmitted sequentially to full adder circuit, which adds up the two digits and finally transmits the result to the B4 block.

Q54. Using flowcharts explain the following,

(a) Decimal multiplication

(b) Decimal division.

Answer :

(a) Decimal Multiplication

Following is a flowchart representing decimal multiplication.

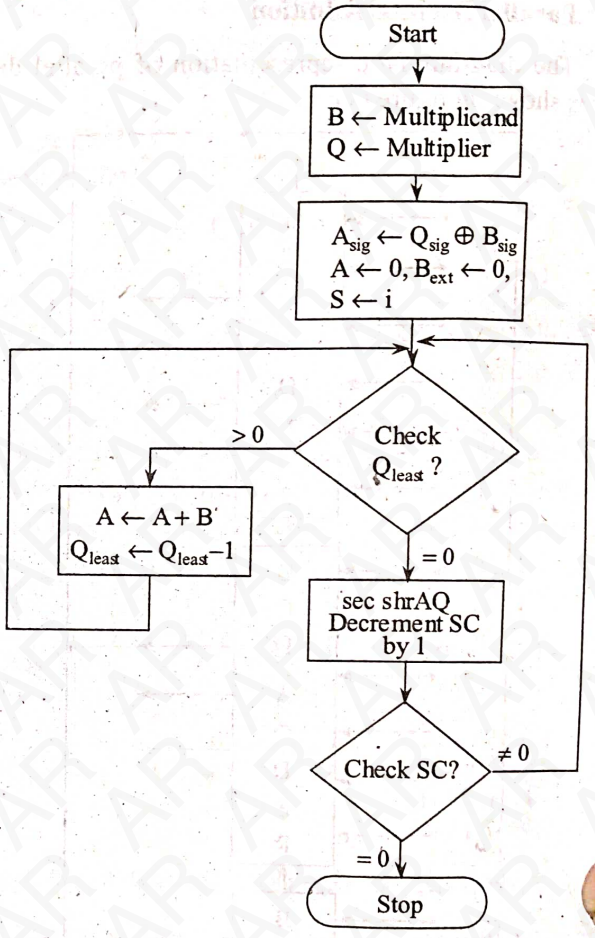


Figure (1): Flowchart for Decimal Multiplication

Various terms used in the flowchart above are as follows,

A, B, Q represent three registers. The terms A_{sig}, B_{sig}, Q_{sig} denotes 1-bit storage locations used to store signs of the bits stored in A, B and Q respectively. The digit i represents the number of digits stored in a multiplier; SC is a sequence counter. Finally, the term Q_{least} represent the least significant bit of Q and B_{ext} stores an overflow bit which results after performing the addition operation on multiplicand and the partial product, during the multiplication operation.

The procedure to perform decimal multiplication is illustrated below,

- Initially, the algorithm begins by storing the multiplicand in register B and the multiplier in register Q.
- Later, the signs of values stored in the registers Q_{sig} and B_{sig} are XORed and the result is directed to register A_{sig}. Register A and B_{ext} are initialized by storing value 0. The value of i is shifted to sequence counter.

- In the next step, the least significant value stored in register Q is checked. Here two possibilities arise

(i) If the value of $Q_{least} = 0$, then the value of AQ is decremented and shifted right and the value of sequence counter is decremented.

(ii) If the value of $Q_{least} \neq 0$, then the values of registers A and B are added whose result is stored A and the value of Q_{least} is decremented by 1.

Moreover the value of Q_{least} is again checked. This process is repeated until the value of Q_{least} turns out to be zero.

Before terminating the process the value of sequence counter is examined. Here again there are two possibilities.

(i) If the value of SC is equal to zero, then the multiplication process is halted by storing the result in AQ.

(ii) If the value of SC is not equal to zero, then the process repeats by examining the Q_{least} value.

(b) Decimal Division

The flowchart depicting decimal division is shown in figure (2).

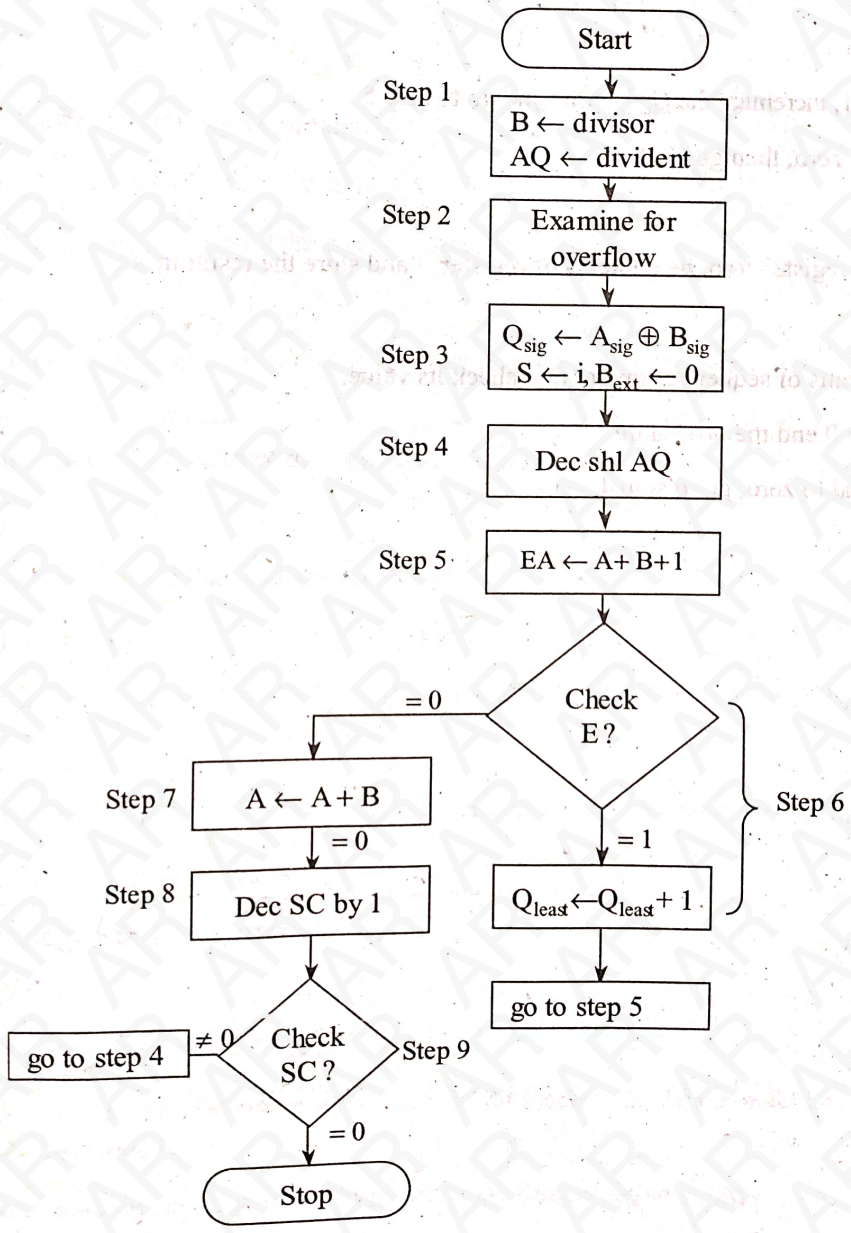


Figure (2): Flowchart for Decimal Division

The steps essential to compute decimal division are as follows.

Step 1

Store the divisor in register B and dividend in register AQ.

Step 2

Check for overflow.

Step 3

Here the signs of A_{sig} and B_{sig} are XORed and the result is stored in Q_{sig}. Later the value of i (which is equal to the number of bits in the quotient) is stored in sequence counter and the register bit B_{ext} is cleared by storing a value 0 in it.

Step 4

The value of AQ is decremented and shifted towards its left.

Step 5

The value of register A is added to B's complement and the result is added to 1 and stored in register EA.

Step 6

Check the value of E.

(i) If it is equal to 1, increment the Q_{least} value and go to step 5.

(ii) If it is equal to zero, then go to step 7.

Step 7

Add the contents of register B to the contents of register A and store the result in A.

Step 8

Decrement the contents of sequence counter and check its value.

(i) If it is equal to 0 end the procedure.

(ii) If it is not equal to zero, go to step 4.

Step 9

Stop.