UNIT - III

Data Representation: Data types, Complements, Fixed Point Representation, Floating Point Representation.

Computer Arithmetic: Addition and subtraction, multiplication Algorithms, Division Algorithms, Floating – point Arithmetic operations. Decimal Arithmetic unit, Decimal Arithmetic operations.

Data Types

- The data types found in the registers of digital computers may be classified as being one of the following categories: (1) numbers used in arithmetic computations, (2) letters of the alphabet used in data processing. and (3) other discrete symbols used for specific purposes.
- All types of data, except binary numbers, are represented in computer registers in binary coded form. This is because registers are made up of flip-flops and flip-flops are two-state devices that can store only l's and O's. The binary number system is the most natural system to use in a digital computer.

Number Systems:

- ➢ A number system of **base**, or **radix**, r is a system that uses distinct symbols for r digits. Numbers are represented by a string of digit symbols.
- ➢ To determine the quantity that the number represents, it is necessary to multiply each digit by an integer power of **r** and then form the sum of all weighted digits.

  For example: The **decimal number system** in everyday use employs the radix 10 system.

  The 10 symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The string of digits 724.5 is interpreted to represent the quantity

  $$7 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 + 5 \times 10^{-1}$$

  that is, 7 hundreds, plus 2 tens, plus 4 units, plus 5 tenths. Every decimal number can be similarly interpreted to find the quantity it represents.

- ➢ The **binary number system** uses the **radix 2**. The two digit symbols used are 0 and 1. The string of digits 101101 is interpreted to represent the quantity

  $$1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 45$$

To distinguish between different radix numbers, the digits will be enclosed in parentheses and the radix of the number inserted as a subscript.

For example, to show the equality between decimal and binary forty-five we will write

$(101101)_2 = (45)_{10}$

- Besides the decimal and binary number systems, the **octal (radix 8) and hexadecimal (radix 16**) are important in digital computer work.
- The eight symbols of the octal system are 0, 1, 2, 3, 4, 5, 6, and 7. The 16 symbols of the hexadecimal system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, 0, E, and F.
- The last six symbols are, unfortunately, identical to the letters of the alphabet and can cause confusion at times. However, this is the convention that has been adopted. When used to represent hexadecimal digits, the symbols A, B, C, D, E, F correspond to the decimal numbers 10, 11, 12, 13, 14, 15, respectively.
- A number in radix r can be converted to the familiar decimal system by forming the sum of the weighted digits.

For example, octal 736.4 is converted to decimal as follows:

$$(736.4)_{\mathbf{8}} = 7 \times 8^2 + 3 \times 8^1 + 6 \times 8^0 + 4 \times 8^{-1}$$

$$= 7 \times 64 + 3 \times 8 + 6 \times 1 + 4/8 = (478.5)_{\mathbf{10}}$$

The equivalent decimal number of hexadecimal **F3** is obtained from the following calculation:

$$(F3)_{\mathbf{16}} = F \times 16^1 + 3 \times 16^0 = 15 \times 16 + 3 = (243)_{\mathbf{10}}$$

Conversion:

Decimal to Binary Conversion:

Conversion from decimal to its equivalent representation in the radix r system is carried out by separating the number into its **integer and fraction parts** and <u>converting each part separately</u>.

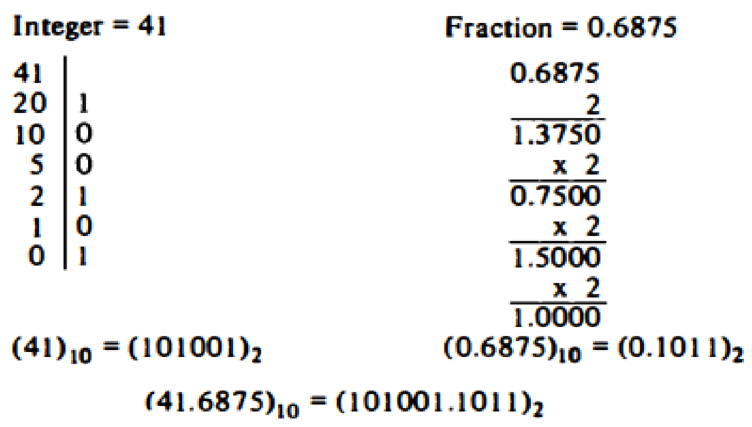
- ✓ The conversion of a decimal integer into a base r representation is done by successive divisions by r and accumulation of the remainders.
- ✓ The conversion of a decimal fraction to radix r representation is accomplished by successive multiplications by r and accumulation of the integer digits so obtained.

Example: The conversion of decimal 41. 6875 into binary

- Figure 3-A demonstrates these procedures
- The conversion of decimal 41.6875 into binary is done by first separating the number into its **integer part 41** and **fraction part .6875**.

- **The integer part** is converted by dividing 41 by r = 2 to give an integer quotient of 20 and a remainder of 1. The quotient is again divided by 2 to give a new quotient and remainder. This process is repeated until the integer quotient becomes 0.
- The fraction part is converted by multiplying it by r = 2 to give an integer and a fraction. The new fraction (without the integer) is multiplied again by 2 to give a new integer and a new fraction. This process is repeated until the fraction part becomes zero or until the number of digits obtained gives the required accuracy.

Figure 3-A: Conversion of decimal 41.6875 into binary.



```
Integer = 41                 Fraction = 0.6875
41 |                              0.6875
20 | 1                                 2
10 | 0                            1.3750
 5 | 0                               x 2
 2 | 1                            0.7500
 1 | 0                               x 2
 0 | 1                            1.5000
                                     x 2
                                  1.0000
```

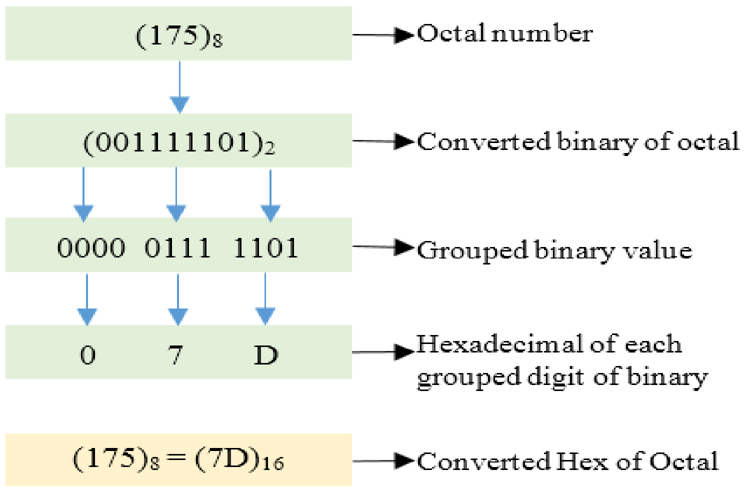
$(41)_{10} = (101001)_2$ $\quad$ $(0.6875)_{10} = (0.1011)_2$

$(41.6875)_{10} = (101001.1011)_2$

Octal and Hexadecimal Numbers:

- The conversion from and to binary, octal, and hexadecimal representation plays an important part in digital computers. Since $2^3 = 8$ and $2^4 = 16$, each octal digit corresponds to <u>three binary digits</u> and each hexadecimal digit corresponds to <u>four binary digits</u>.
- The conversion from binary to octal is easily accomplished by partitioning the binary number into groups of three bits each. The corresponding octal digit is then assigned to each group of bits and the string of digits so obtained gives the octal equivalent of the binary number.

Example:



- The corresponding octal digit for each group of three bits is easily remembered after studying the first eight entries listed in Table 3-1

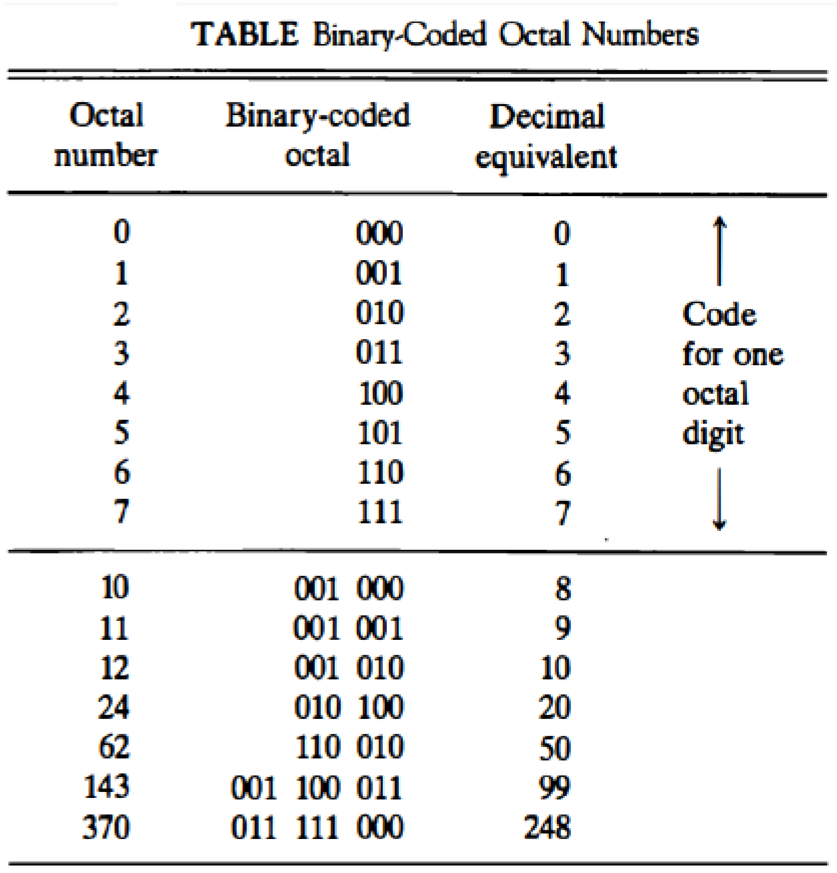


TABLE Binary-Coded Octal Numbers

Octal number	Binary-coded octal	Decimal equivalent	
0	000	0	↑
1	001	1	
2	010	2	Code
3	011	3	for one
4	100	4	octal
5	101	5	digit
6	110	6	
7	111	7	↓
10	001 000	8	
11	001 001	9	
12	001 010	10	
24	010 100	20	
62	110 010	50	
143	001 100 011	99	
370	011 111 000	248	

- The correspondence between a hexadecimal digit and its equivalent 4-bit code can be found in the first 16 entries of Table 3-2.

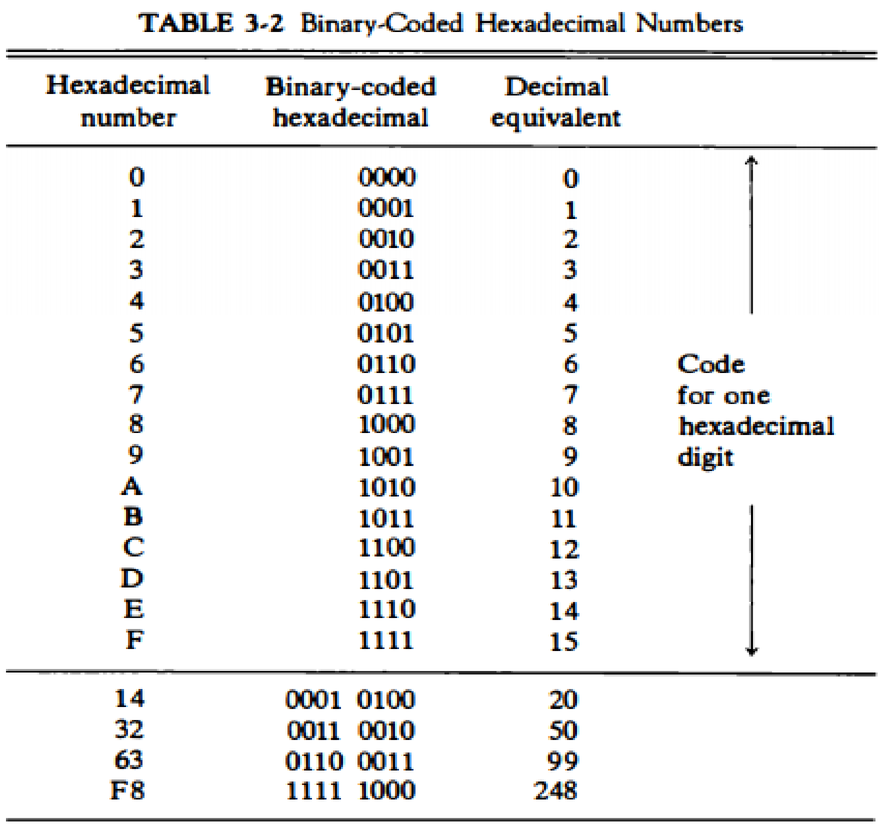


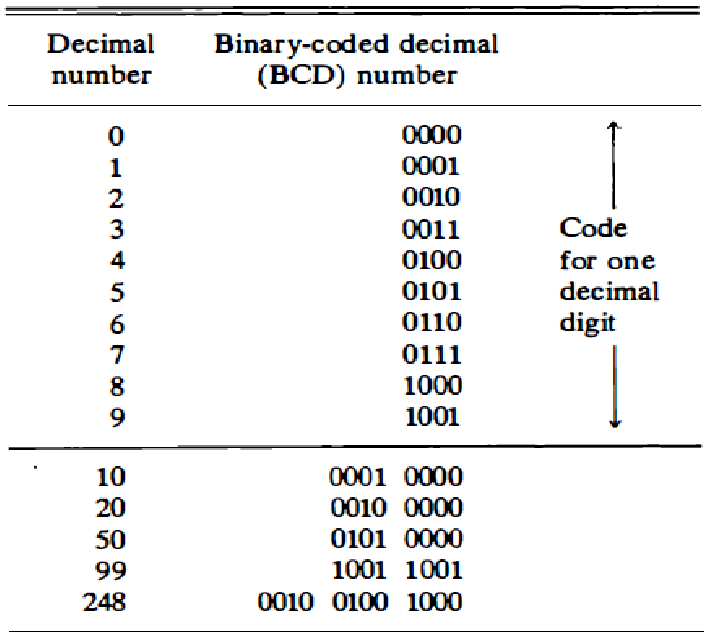
TABLE 3-2 Binary-Coded Hexadecimal Numbers

Hexadecimal number	Binary-coded hexadecimal	Decimal equivalent	
0	0000	0	↑
1	0001	1	
2	0010	2	
3	0011	3	
4	0100	4	
5	0101	5	
6	0110	6	Code
7	0111	7	for one
8	1000	8	hexadecimal
9	1001	9	digit
A	1010	10	
B	1011	11	
C	1100	12	
D	1101	13	
E	1110	14	
F	1111	15	↓
14	0001 0100	20	
32	0011 0010	50	
63	0110 0011	99	
F8	1111 1000	248	

Binary-Coded Decimal (BCD):

- It is very important to understand the difference between the conversion of decimal numbers into binary and the binary coding of decimal numbers.
- For example, when converted to a binary number, the decimal number 99 is represented by the string of bits 1100011, but when represented in BCD, it becomes 1001 1001.
- The only difference between a decimal number represented by the familiar digit symbols 0, 1, 2, . .. , 9 and the BCD symbols 0001, 0010, . . . , 1001 is in the symbols used to represent the digits-the number itself is exactly the same.
- A few decimal numbers and their representation in BCD are listed in Table 3-3.

TABLE 3-3 Binary-Coded Decimal (BCD) Numbers



Decimal number	Binary-coded decimal (BCD) number	
0	0000	
1	0001	
2	0010	
3	0011	Code
4	0100	for one
5	0101	decimal
6	0110	digit
7	0111	
8	1000	
9	1001	
10	0001 0000	
20	0010 0000	
50	0101 0000	
99	1001 1001	
248	0010 0100 1000	

Complements

Complements are used in digital computers for simplifying the subtraction operation and for logical manipulation. There are two types of complements for each base r system: the r's complement and the (r - l)'s complement.

9’s complement:

- Given a number N in base **r** having n digits, the (**r - 1)**'s complement of N is defined as (r' - 1) - N.
- For decimal numbers r = 10 and r - 1= 9, so the 9's complement of N is (10^n - 1) - N. Now, 10^n represents a number that consists of a single 1 followed by n 0's. 10^n - 1 is a number represented by n 9's.
- For example, with n = 4 we have 10^4 = 10000 and 10^4 - 1 = 9999. It follows that the 9' s complement of a decimal number is obtained by subtracting each digit from 9.
- **For example:**

  9's complement of 546700 is 999999 - 546700 = 453299 .

  9's complement of 12389 is 99999 - 12389 = 87610

1's complement:

- **For example**: The I's complement of 10110011 is 0100110 and the 1' s complement of 0001111 is 1110000.

10's complement:

It obtained by adding 1 to the 9' s complement value

- **For example:**

The 10's complement of 246700 is 753300 and is obtained.
9' s complement value 999999 – 246700 = 753299
+1

The 10's complement = 753300

2's complement:

- The 2's complement can be formed by leaving all least significant 0's and the first 1 unchanged, and then replacing 1's by 0's and 0's by 1's in all other higher significant bits.
- **Example:**

The 2's complement of 1101100 is 0010100 and is obtained by leaving the two low-order 0's and the first 1 unchanged, and then replacing 1's by 0's and 0's by 1's in the other four most significant bit.

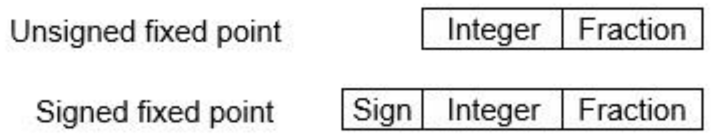
There are various types of number representation techniques for digital number representation, for example: Binary number system, octal number system, decimal number system, and hexadecimal number system etc. But Binary number system is most relevant and popular for representing numbers in digital computer system.

- ❖ There are two major approaches to store real numbers (i.e., numbers with fractional component) in modern computing.
- ❖ These are (i) Fixed Point Notation and (ii) Floating Point Notation.
- ❖ In fixed point notation, there are a fixed number of digits after the decimal point, whereas floating point number allows for a varying number of digits after the decimal point.

Fixed-Point Representation –

- In ordinary arithmetic, a negative number is indicated by a minus sign and a positive number by a plus sign. Because of hardware limitations, computers must represent everything with 1's and 0's, including the sign of a number.
- In addition to the sign, a number may have a binary (or decimal) point. The position of the binary point is needed to represent fractions, integers, or mixed integer-fraction numbers.
- The representation of the binary point in a register is complicated by the fact that it is characterized by a position in the register.
- There are two ways of specifying the position of the binary point in a register: by giving it a fixed position or by employing a floating-point representation. The fixed-point method assumes that the binary point is <u>always fixed in one position</u>.

This representation has fixed number of bits for integer part and for fractional part. There are three parts of a fixed-point number representation: the sign field, integer field, and fractional field.



We can represent these numbers using:

- Signed representation:
- 1’s complement representation:
- 2’s complementation representation:

❖ 2’s complementation representation is preferred in computer system because of unambiguous property and easier for arithmetic operations.

Example:1 – Represent of fixed Point representations unsigned binary number 0110110 using 4 integer bits and 3 fractional.

Solution: 0110110

0110 . 110

Example:2 – Represent $(-7.5)_{10}$ using 8-bit binary representation with 4 digits as integer and 4 as fractional bits.

Solution: $(-7.5)_{10}$

$(111.1)_2$ -------> Binary form

0111 . 1000 -------> converting into 8-bit Binary form

To convert into negative number

Finding 2’s complement **0 1 1 1 . 1 0 0 0**

1 0 0 0 . 0 1 1 1

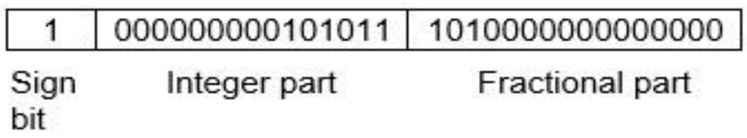
+1

We wil get 1000 . 1000

$(-7.5)_{10} = (\mathbf{1000\,.\,1000})_2$

Example:3 −Assume number is using 32-bit format which reserve 1 bit for the sign, 15 bits for the integer part and 16 bits for the fractional part.

- Then, -43.625 is represented as following:



1	000000000101011	1010000000000000
Sign bit	Integer part	Fractional part

Where, 0 is used to represent + and 1 is used to represent -. 000000000101011 is 15 bit binary value for decimal 43 and 1010000000000000 is 16 bit binary value for fractional 0.625.

Example:4 –Convert 7.75 decimal number to fixed point binary number.

Solution: 111.110

Floating-Point Representation

Floating-point representation is similar in concept to scientific notation.

The floating number representation of a number has two part: the first part represents a signed fixed point number called mantissa. The second part of designates the position of the decimal (or binary) point and is called the exponent. The fixed point mantissa may be fraction or an integer.

Floating -point is always interpreted to represent a number in the following form:

$$\pm M \times B^{E}$$

– M = mantissa

– B = base

– E = exponent

The Normalized Mantissa –

The mantissa is part of a number in scientific notation or a floating-point number, consisting of its significant digits. Here we have only 2 digits, i.e. 0 and 1. So a normalized mantissa is one with only one 1 to the left of the decimal.

Example:1: , Write 273_{10} in scientific notation:

$$273 = 2.73 \times 10^{2}$$

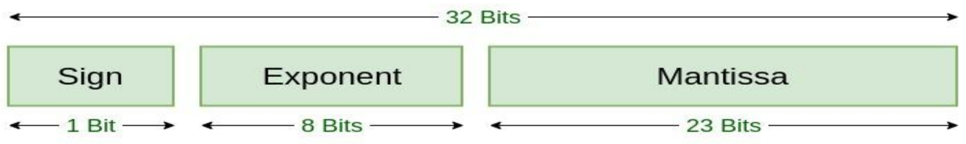
In the example, M = 2.73, B = 10, and E = 2

Modern computers adopt IEEE 754 standard (Institute of Electrical and Electronics Engineers) for representing floating-point numbers.

There are two representation schemes:

1. 32-bit single-precision
2. 64-bit double-precision.

1. single-precision (32-BIT) :



Example:1- Represent the value 228_{10} using a 32-bit floating point representation

1. Convert decimal to binary

228_{10} = 111001002

2. Write the number in "binary scientific notation":

$11100100_2 = 1.110012 \times 2^7$

3. Fill in each field of the 32-bit floating point number:

– The sign bit is positive (0)

– The 8 exponent bits represent the value 7

– The remaining 23 bits are the mantissa

NOTE: First bit of the mantissa is always 1: So, no need to store it: implicit leading 1. Store just fraction bits in 23-bit field

1-Bit	8-bit	23-bit
0	**00000111**	**110 0100 0000 0000 0000 0000**
Sign	**Exponent**	**Fraction**

Biased exponent Method:

Biased exponent: bias = 127 (01111111_2)

– Biased exponent = bias + exponent

– Exponent of 7 is stored as: 127 + 7 = 134 = 10000110_2

The IEEE 754 32-bit floating-point representation of 228_{10}

1-Bit	8-bit	23-bit
0	**10000110**	**110 0100 0000 0000 0000 0000**
Sign	**Biased Exponent**	**Fraction**

in hexadecimal: 0x43640000

Example:2- Represent the value -58.25$_{10}$ using a 32-bit floating point representation

1. Convert decimal to binary:

$58.25_{10} = 111010.01_2$

2. Write in binary scientific notation:

1.1101001×2^5

3. Fill in fields:

Sign bit: 1 (negative)

8 exponent bits: (127 + 5) = 132 = 10000100_2

23 fraction bits: 110 1001 0000 0000 0000 0000

The IEEE 754 32-bit floating-point representation of -58.25$_{10}$

1-Bit	8-bit	23-bit
1	**10000100**	110 1001 0000 0000 0000 0000
Sign	**Biased Exponent**	**Fraction**

in hexadecimal: 0xC2690000

Example:3 - Represent the value (1259.125)$_{10}$ using a 32-bit floating point representation

1. Convert decimal to binary:

$1259.125_{10} = 10011100011.001_2$

2. Write in binary scientific notation:

$1.0011100011001 \times 2^{10}$

3. Fill in fields:

Sign bit: 0 (positive)

8 exponent bits: (127 + 10) = 137 = 10001001_2

23 fraction bits: 00111000110010000000000

1-Bit	8-bit	23-bit
0	10001001	00111000110010000000000
Sign	**Biased Exponent**	**Fraction**

Example:4 - Represent the value $(263.3)_{10}$ using a 32-bit floating point representation

1. Convert decimal to binary:

$263.3 = 100000111.01\ 0011\ 0011\ 0011\ldots_2$

2. Write in binary scientific notation:

$1.\ 0000011101\ 0011\ 0011\ 0011\ldots \times 2^{8}$

3. Fill in fields:

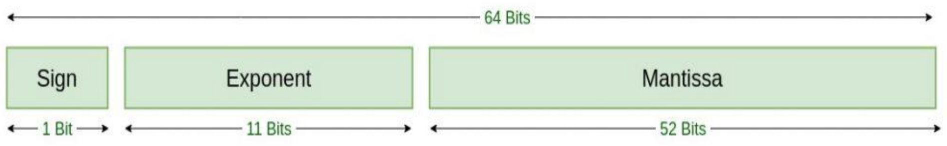
Sign bit: 0 (positive)

8 exponent bits: $(127 + 8) = 135 = 10000111_2$

23 fraction bits: 0000011101 0011 0011 0011

1-Bit	8-bit	23-bit
0	10000111	0000011101 0011 0011 00110
Sign	**Biased Exponent**	**Fraction**

2. **Double-precision (62-BIT)** :



Example:1 - Represent the value $(1259.125)_{10}$ using a 64-bit floating point representation

1. Convert decimal to binary:

$1259.125_{10} = 10011100011.001_2$

2. Write in binary scientific notation:

$1.0011100011001 \times 2^{10}$

3. Fill in fields:

Sign bit: 0 (positive)

11 exponent bits: $(1023 + 10) = 1033 = 10001001_2$

52 fraction bits: 00111000110010000000000

1-Bit	11-bit	52-bit
0	10000001001	00111000110010000000000------(upto 52 bits)
Sign	**Biased Exponent**	**Fraction**

Arithmetic Addition:

- The addition of two numbers in the signed-magnitude system follows the rules of ordinary arithmetic. If the signs are the same, we add the two magnitudes and give the sum the common sign.
- If the signs are different, we subtract the smaller magnitude from the larger and give the result the sign of the larger magnitude.

For example, (+ 25) + (- 37) = - (37 - 25) = - 12

- **Numerical examples** for addition are shown below. Note that negative numbers must initially be in 2’s complement and that if the sum obtained after the addition is negative, it is in 2's complement form.

```
+6      00000110                                -6     11111010
+13     00001101                              +13    00001101
        -----------------                             -----------------
+19     00010011                              +7    000001 11
      --------------------                          ---------------------
+6   00000110                              - 6     11111010
- 13 11110011                            - 13     11110011
--------------------                         ----------------------
-7   1111 1001                             -19   11101101
-------------------                          ---------------------
```

- The complement form of representing negative numbers is unfamiliar to people used to the signed-magnitude system.
- To determine the value of a negative number when in signed-2's complement, it is necessary to convert it to a positive number to place it in a more familiar form.
- **For example,** the signed binary number 1111 1001 is negative because the leftmost bit is I. Its 2' s complement is 00000111, which is the binary equivalent of +7. We therefore recognize the original negative number to be equal to - 7

Arithmetic Subtraction:

- Subtraction of two signed binary numbers when negative numbers are in 2' s complement form is very simple and can be stated as follows: Take the 2's complement of the subtrahend (including the sign bit) and add it to the minuend (including the sign bit).
- A carry out of the sign bit position is discarded

.

This is demonstrated by the following relationship:

$$(\pm A) - (+B) = (\pm A) + (-B)$$
$$(\pm A) - (-B) = (\pm A) + (+B)$$

- But changing a positive number t o a negative number i s easily done b y taking its 2's complement.
- The reverse is also true because the complement of a negative number in complement form produces the equivalent positive number.

Consider the subtraction of (-6) - (- 13) = +7.

In binary with eight bits this is written as

11111010 - 11110011 .

The subtraction is changed to addition by taking the 2's complement of the subtrahend (- 13) to give (+ 13).

In binary this is 1111 1010 + 00001101 = 100000111 . Removing the end carry, we obtain the correct answer 00000111 (+ 7).

Computer Arithmetic

Addition and Subtraction:

➢ There are three ways of representing negative fixed-point binary numbers:
  - signed-magnitude
  - signed-l's complement or signed-2's complement.

➢ Most computers use the signed-2's complement representation when performing arithmetic operations with integers.

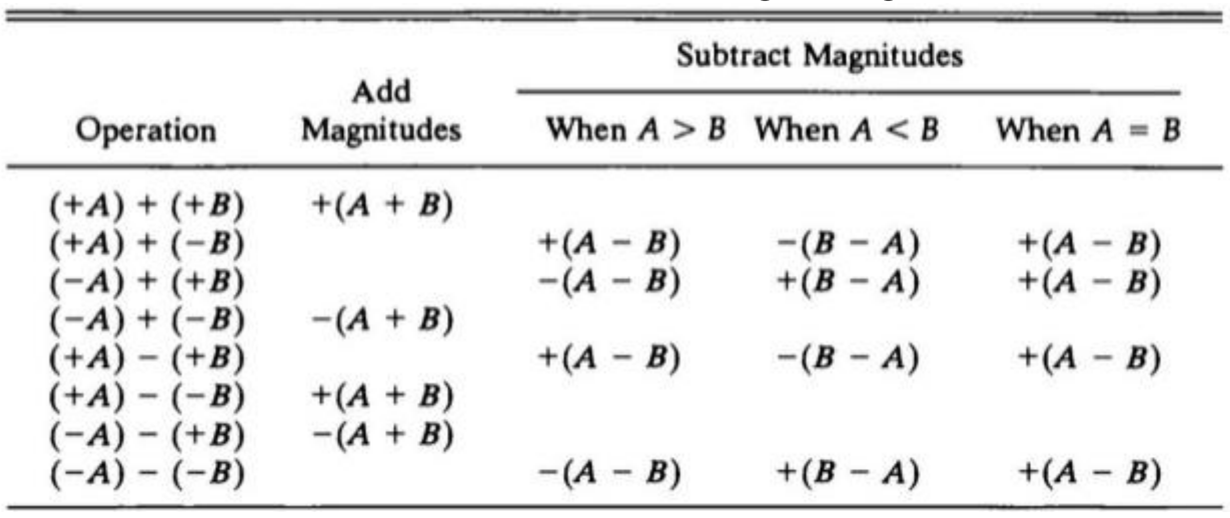
➢ For floating-point operations, most computers use the signed-magnitude representation for the mantissa.

Here we discuss the addition and subtraction algorithms for data represented in signed-magnitude and again for data represented in signed-2's complement.

Addition and Subtraction with Signed-Magnitude Data:

- We designate the magnitude of the two numbers by A and B. When the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed.
- These conditions are listed in the first column of Table 3-4. The other columns in the table show the actual operation to be performed with the magnitude of the numbers. The last column is needed to prevent a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0.

Table-3-4: Addition and Subtraction of Signed-Magnitude Numbers



		Subtract Magnitudes		
Operation	Add Magnitudes	When $A > B$	When $A < B$	When $A = B$
$(+A) + (+B)$	$+(A + B)$			
$(+A) + (-B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(-A) + (+B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$
$(-A) + (-B)$	$-(A + B)$			
$(+A) - (+B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(+A) - (-B)$	$+(A + B)$			
$(-A) - (+B)$	$-(A + B)$			
$(-A) - (-B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$

The algorithms for addition and subtraction are derived from the table and can be stated as follows (the words inside parentheses should be used for the subtraction algorithm)

Addition algorithm: when the signs of A and B are identical, add the two magnitudes and attach the sign of A to the result. When the signs of A and B are different, compare the magnitudes and subtract the smaller number from the larger.

- Choose the sign of the result to be the same as A if A > B or the complement of the sign of A if A < B.
- If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

Subtraction algorithm: when the signs of A and B are different, add the two magnitudes and attach the sign of A to the result. When the signs of A and B are identical, compare the magnitudes and subtract the smaller number from the larger.

- Choose the sign of the result to be the same as A if A > B or the complement of the sign of A if A < B. If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

Hardware Algorithm:

The flowchart for the hardware algorithm is presented in Fig. 3-B.

Step1: The two signs A_S, and B_S, are compared by an exclusive-OR gate. If the output of the gate is 0, the signs are identical; if it is 1, the signs are different.

Step2: For an add operation, identical signs dictate that the magnitudes be added. For a subtract operation, different signs dictate that the magnitudes be added.

Step3:

- The magnitudes are <u>added</u> with a micro operation $EA \leftarrow A + B$. where EA is a register that Combines E and A. The carry in E after the addition constitutes an overflow if it is equal to 1. The value of E is transferred into the add-overflow flip-flop AVF.
- The two magnitudes are <u>subtracted</u> if the signs are different for an add operation or identical for a subtract operation. The magnitudes are subtracted by adding A to the 2's complement of B. No overflow can occur if the numbers are subtracted so AVF is cleared to 0.

Step4:

- If $E \leftarrow 1$ indicates that $A >= B$ and the number in A is the correct result. If this number is zero, the sign A_S, must be made positive to avoid a negative zero.
- If $E \leftarrow 0$ indicates that $A < B$. For this case it is necessary to take the 2's complement of the value in A. This operation can be done with one micro operation $A \leftarrow -A + 1$.
- When $A < B$, the sign of the result is the complement of the original sign of A. It is then necessary to complement A_S, to obtain the correct sign. The final result is found in register A and its sign in A_S.

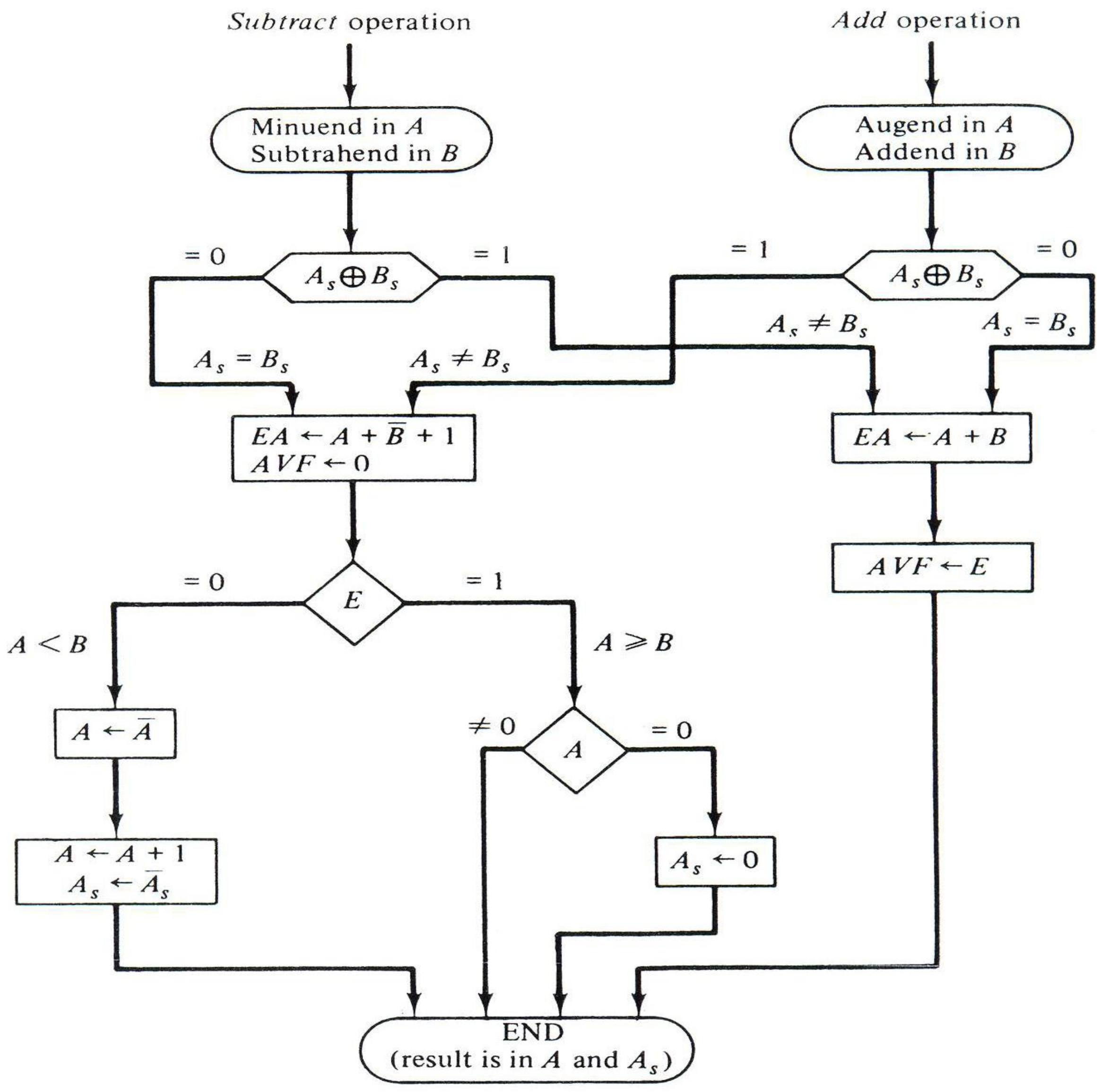


Figure 3-B: Flow chart for add and subtract operations.

Addition and Subtraction with Signed-2's Complement Data:

- The leftmost bit of a binary number represents the sign bit: 0 for positive and 1 for negative. If the sign bit is 1, the entire number is represented in 2' s complement form.
- Thus + 33 is represented as 00100001 and - 33 as 1101 1 1 1 1 . Note that 11011111 is the 2's complement of 00100001, and vice versa.

Hardware Algorithm:

The algorithm for adding and subtracting two binary numbers in signed- 2’s complement representation is shown in the flowchart of Fig. 3-C.

Step 1: The sum is obtained by adding the contents of AC and BR (including their sign bits and here we name the A register AC (accumulator) and the B register BR).

Step 2: The overflow bit V is set to 1 if the exclusive-OR of the last two carries is 1, and it is cleared to 0 otherwise.

Step 3: The subtraction operation is accomplished by adding the content of AC to the 2's complement of BR .Taking the 2's complement of BR has the effect of changing a positive number to negative, and vice versa.

Step 4: An overflow must be checked during this operation because the two numbers added could have the same sign. The programmer must realize that if an overflow occurs, there will be an erroneous result in the AC register.

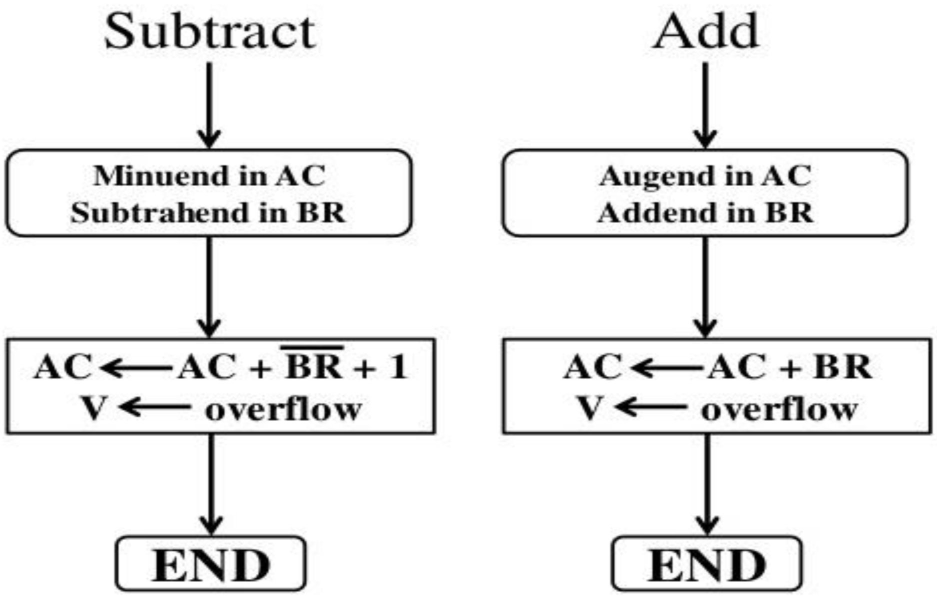


Fig 3-C: Algorithm for adding and subtracting numbers in signed 2's complement representation.

- Comparing this algorithm with its signed-magnitude counterpart, we note that it is much simpler to add and subtract numbers if negative numbers are maintained in signed-2' s complement representation.
- For this reason most computers adopt this representation over the more familiar signed-magnitude

Multiplication Algorithm:

- Multiplication of two fixed-point binary numbers in signed-magnitude representation is done with paper and pencil by a process of successive shift and add operations.

This process is best illustrated with a numerical example.

```
23          1 0 1 1 1        Multiplicand
19        X 1 0 0 1 1        Multiplier
--------------------------
               10111
              10111
             00000     +
            00000
           10111
-----------------------------
437  1101 10101              Product
```

- The process consists of looking at successive bits of the multiplier, least significant bit first. If the multiplier bit is a 1, the multiplicand is copied down; otherwise, zeros are copied down.
- The numbers copied down in successive lines are shifted one position to the left from the previous number. Finally, the numbers are added and their sum forms the product.
- The sign o f the product is determined from the signs of the multiplicand and multiplier. If they are alike, the sign of the product is positive. If they are unlike, the sign of the product is negative.

Hardware Algorithm:

Figure 3-D is a flowchart of the hardware multiply algorithm.

Step 1:

- Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in B_S, and Q_S, respectively.
- The signs are compared, and both A and Q are set to correspond to the sign of the product since a double-length product will be stored in registers A and Q.

Step 2: Registers A and E are cleared and the sequence counter SC is set to a number equal to the number of bits of the multiplier.

Step 3: After the initialization, the low-order bit of the multiplier in Q_n, is tested. If it is a 1, the multiplicand in B is added to the present partial product in A .If it is a 0, nothing is done.

Step 4: Register EAQ is then shifted once to the right to form the new partial product. The sequence counter is decremented by 1 and its new value checked. If it is not equal to zero, the process is repeated and a new partial product is formed. The process stops when SC = 0.

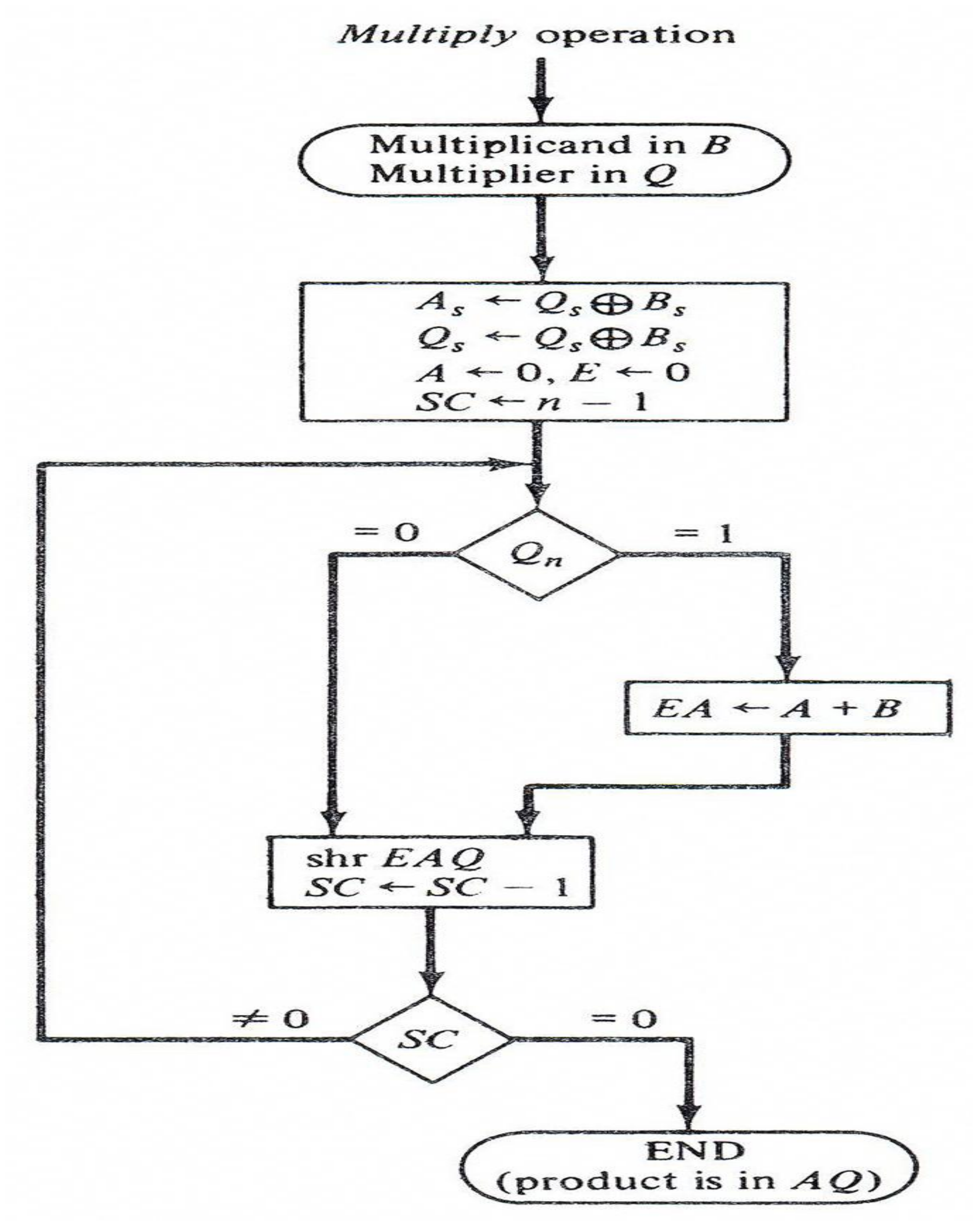
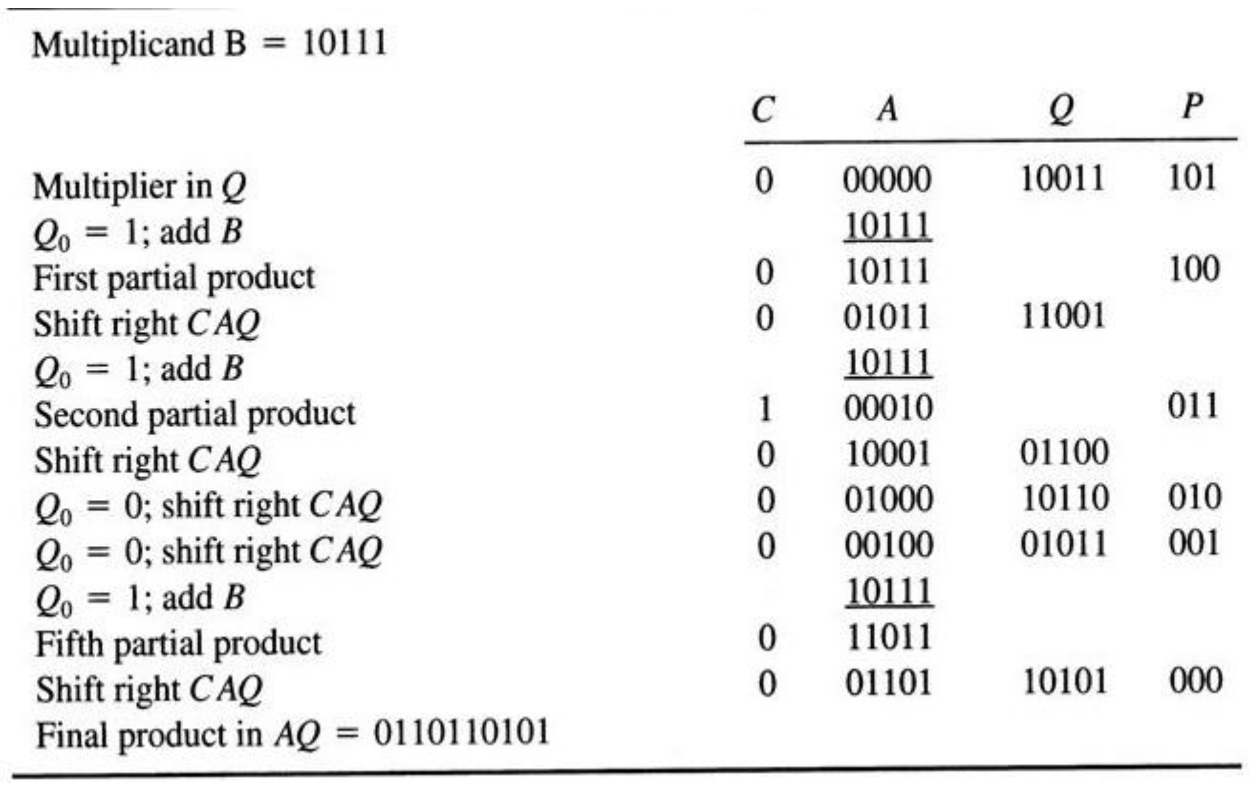


Fig 3-D: Flowchart for multiply operation

NOTE: The partial product formed in A is shifted into Q one bit at a time and eventually replaces the multiplier. The final product is available in both A and Q, with A holding the most significant bits and Q holding the least significant bits

Numerical Example : The previous numerical example is repeated in Table 3-E to clarify the hardware multiplication process. The procedure follows the steps outlined in the flowchart. (here P refers the SC and C Reffered as E)

Table 3-E: Numerical Example for Binary Multiplier



Multiplicand B = 10111

	C	*A*	*Q*	*P*
Multiplier in *Q*	0	00000	10011	101
$Q_0 = 1$; add *B*		$\underline{10111}$		
First partial product	0	10111		100
Shift right *CAQ*	0	01011	11001	
$Q_0 = 1$; add *B*		$\underline{10111}$		
Second partial product	1	00010		011
Shift right *CAQ*	0	10001	01100	
$Q_0 = 0$; shift right *CAQ*	0	01000	10110	010
$Q_0 = 0$; shift right *CAQ*	0	00100	01011	001
$Q_0 = 1$; add *B*		$\underline{10111}$		
Fifth partial product	0	11011		
Shift right *CAQ*	0	01101	10101	000

Final product in *AQ* = 0110110101

Booth Multiplication Algorithm:

- Booth algorithm gives a procedure for multiplying binary integers in signed-2's complement representation.

- It operates on the fact that strings of 0's in the multiplier require no addition but just shifting, and a string of 1's in the multiplier from bit weight 2^K to weight 2^m can be treated as $2^{K+1} - 2^m$.
- **For example**, the binary number 001110 (+ 14) has a string of 1's from 2^3 to 2^1 (k = 3, m = 1).

  The number can be represented as $2^{K+1} - 2^m = 2^4 - 2^1 = 16 - 2 = 14$.

  Therefore, the multiplication M **x** 14, where M is the multiplicand and 14 the multiplier, can be done as M **x** 2^4 - M **x** 2^1 .

  Thus the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.

- As in all multiplication schemes, Booth algorithm requires examination of the multiplier bits and shifting of the partial product.
- Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial product, or left unchanged according to the following rules:
  1. The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier.
  2. The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous 1) in a string of 0's in the multiplier.
  3. The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

The Hardware Implementation:

The registers are AC, BR, and QR.

Step 1: The multiplicand in BR and Q_n, designates the least significant bit of the multiplier in register Q_R . An extra flip-flop Q_{n+1} is appended to Q_R to facilitate a double bit inspection of the Multiplier.

Step 2: AC and the appended bit Q_{n+1} are initially cleared to 0 and the sequence counter SC is set to a number **n** equal to the number of bits in the multiplier.

Step 3: The two bits of the multiplier in Q_n, and Q_{n+1} are inspected:-

- If the two bits are equal to 10, it means that the first 1 in a string of 1' s has been encountered. This requires a subtraction of the multiplicand from the partial product in AC .
- If the two bits are equal to 01, it means that the first 0 in a string of 0' s has been encountered. This requires the addition of the multiplicand to the partial product in AC .
- When the two bits are equal, the partial product does not change

Step 4: The next step is to shift right the partial product and the multiplier (including bit Q_{n+1}). This is an arithmetic shift right (**ashr**) operation which shifts AC and QR to the right and leaves the sign bit in AC unchanged .The sequence counter is decremented and the computational loop is repeated n times.

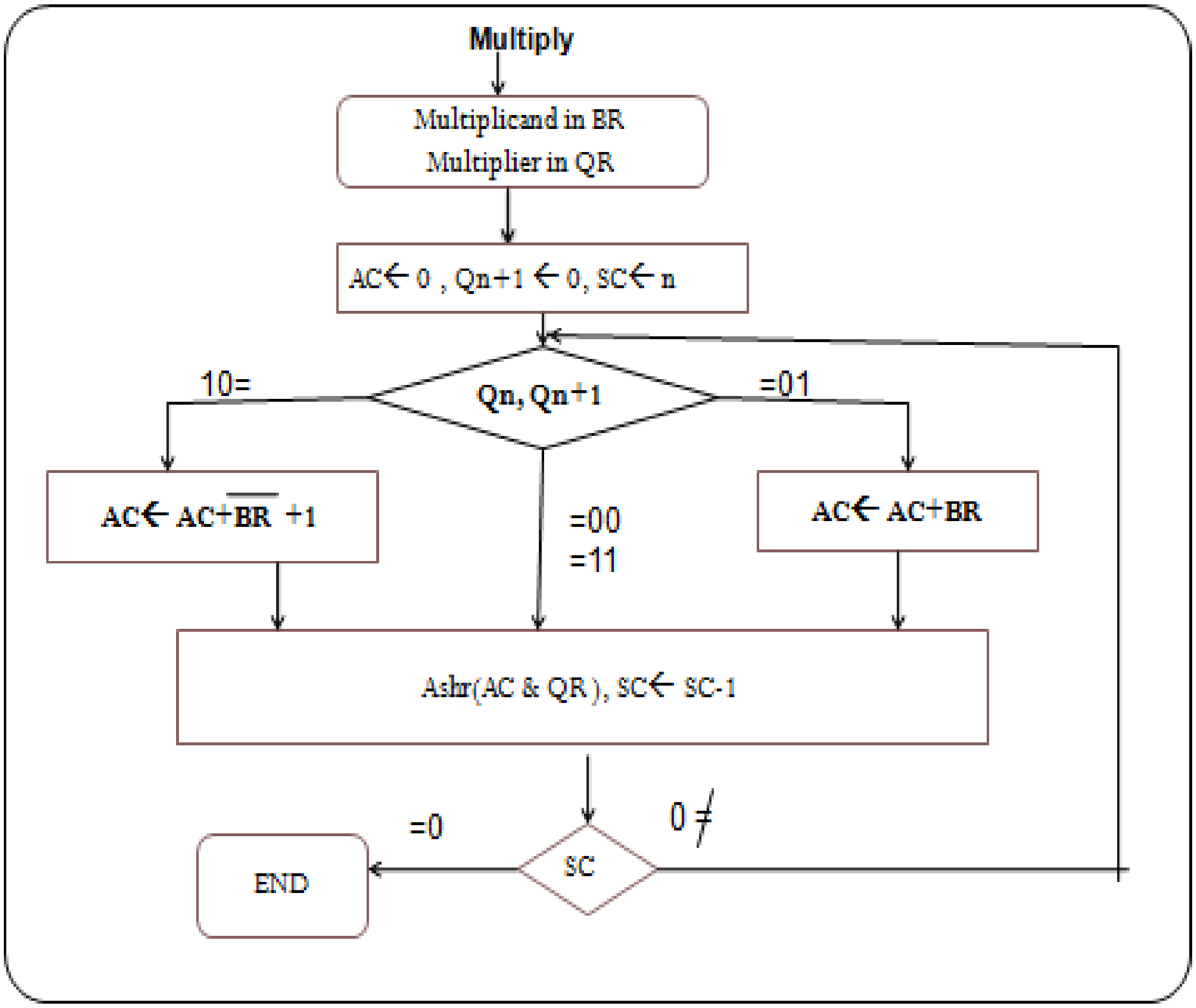


Figure 3-F: Booth algorithm for multiplication o f signed 2's complement numbers.

A numerical Example of Booth algorithm:

Note: Actually value of 9 is 1001 but here we are using 5 bit register so 01001. To get -9. I will do 2’s complement i.e 10111

Example 1: **(-9)** → 1 0 1 1 1 Multiplicand
(-13) → 1 0 0 1 1 Multiplier

Q_n Q_{n+1}	BR=10111 (-BR)+1 =01001	AC	QR	Qn+1	SC
1 0	Initial	00000	10011	0	101
	Subtract BR	01001			

		01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	10111			

		11001			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract BR	01001			

		00111			
	ashr	00011	10101	1	000

❖ Here the final product stored in AC and QR is 0001110101 is **+117**

Example 2: Multiply (+15) X (+13)

Consider multiply +15 and +13. Binary representation of +15 is 01111. i.e BR= 01111 (+15)

-15 is (-BR)+1 i.e 10001 (-15)

+13 is QR i.e 01101 (+13).

- In 1st iteration least significant bit of Q0 is 1 and Q_{n+1} is 0 so subtract BR from AC and store in AC. Right shift QR and AC and decrease SC by 1.

Qn	Q_{n+1}		AC	QR	Q_{n+1}	SC
1	0	Initial Subtract BR	00000 10001	01101	0	101
			10001			
		ashr	11000	10110	1	100

- In 2nd iteration least significant bit of Q_0 is 0 and Q_{n+1} is 1 so add BR to AC and store in AC. Right shift QR and AC and decrease SC by 1.

Qn	**Q_{n+1}**		**AC**	**QR**	**Q_{n+1}**	**SC**
0	1	Add BR	01111			
			00111			
		ashr	00011	11011	0	011

- In 3rd iteration least significant bit of Q0 is 1 and Q_{n+1} is 0 so subtract BR from AC and store in AC. Right shift QR and AC and decrease SC by 1.

Qn	**Q_{n+1}**		**AC**	**QR**	**Q_{n+1}**	**SC**
1	0	Subtract BR	10001			
			10100			
		ashr	11010	01101	1	010

- In 4th iteration least significant bit of Q0 is 1 and Q_{n+1} is 1 so, Right shift QR and AC and decrease SC by 1.

Qn	**Q_{n+1}**		**AC**	**QR**	**Q_{n+1}**	**SC**
1	1	ashr	11101	00110	1	001

- In 5th iteration least significant bit of Q_0 is 0 and Q_{n+1} is 1 so add BR to AC and store in AC. Right shift QR and AC and decrease SC by 1.

Qn	**Q_{n+1}**		**AC**	**QR**	**Q_{n+1}**	**SC**
0	1	Add BR	01111			
			01100			
		ashr	00110	00011	0	000

❖ When (+15) multiplied by (+13) gives +195 = $(0011000011)_2$

Example 3: Multiply (+15) X (-13)

Consider multiply +15 and +13. Binary representation of +15 is 01111. i.e BR= 01111 (+15)
-15 is (-BR)+1 i.e 10001 (-15)
-13 is QR (13 binary representation is 01101 and -13 is 10011). Now **10011 in QR**.

- In 1st iteration least significant bit of Q0 is 1 and Q_{n+1} is 0 so subtract BR from AC and store in AC. Right shift QR and AC and decrease SC by 1.

Qn	**Q_{n+1}**		**AC**	**QR**	**Q_{n+1}**	**SC**
1	0	Initial Subtract BR	00000 10001	10011	0	101
			10001			
		ashr	11000	11001	1	100

- In 2nd iteration least significant bit of Q_0 is 1 and Q_{n+1} is 1 so, Right shift QR and AC and decrease SC by 1.

Qn	**Q_{n+1}**		**AC**	**QR**	**Q_{n+1}**	**SC**
1	1	ashr	11100	01100	1	011

- In 3rd iteration least significant bit of Q0 is 0 and Q_{n+1} is 1 so add BR to AC and store in AC. Right shift QR and AC and decrease SC by 1.

Qn	**Q_{n+1}**		**AC**	**QR**	**Q_{n+1}**	**SC**
0	1	Add BR	01111			
			01011			
		ashr	00101	10110	0	010

- In 4th iteration least significant bit of Q0 is 1 and Q_{n+1} is 0 so , Right shift QR and AC and decrease SC by 1.

Qn	**Q_{n+1}**		**AC**	**QR**	**Q_{n+1}**	**SC**
0	0	ashr	00010	11011	0	001

- In 5th iteration least significant bit of Q_0 is 1 and Q_{n+1} is 0 so subtract BR from AC and store in AC. Right shift QR and AC and decrease SC by 1.

Qn	**Q_{n+1}**		**AC**	**QR**	**Q_{n+1}**	**SC**
1	0	Subtract BR	10001			
			10011			
		ashr	11001	11101	1	000

❖ When (+15) multiplied by (-13) gives -195 = $(1100111101)_2$

Division Algorithm:

Division of two fixed-point binary numbers in signed-magnitude representation is done with paper and pencil by a process of successive compare, shift, and subtract operations.

Numerical example:

- The divisor B consists of five bits and the dividend A consists of ten bits.
- The five most significant bits of the dividend are compared with the divisor.
- Since the 5-bit number is smaller than B, we try again by taking the six most significant bits of A and compare this number with B.
- The 6-bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend.
- The divisor is then shifted once to the right and subtracted from the dividend. The difference is called a partial remainder because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder.
- The process is continued by comparing a partial remainder with the divisor
  - ➢ If the partial remainder is greater than or equal to the divisor, the quotient bit is equal to 1 . The divisor is then shifted right and subtracted from the partial remainder.
  - ➢ If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction is needed. The divisor is shifted once to the right in any case. Note that the result gives both a quotient and a remainder.

Hardware Implementation for Signed-Magnitude Data

- ➢ When the division is implemented in a digital computer, it is convenient to change the process slightly. Instead of shifting the divisor to the right, the dividend, or partial remainder, is shifted to the left, thus leaving the two numbers in the required relative position.
- ➢ Subtraction may be achieved by adding A to the 2's complement of B. The information about the relative magnitudes is then available from the end-carry**.**

Example:

- ❖ The divisor is stored in the B register and the double-length dividend is stored in registers A and Q.
- ❖ The dividend is shifted to the left and the divisor is subtracted by adding its 2' s complement value. The information about the relative magnitude is available in E.
  - ➢ If E = 1, it signifies that A >=B. A quotient bit 1 is inserted into Q, and the partial remainder is shifted to the left to repeat the process.
  - ➢ If E = 0, it signifies that A < B so the quotient in Q, remains a 0 (inserted during the shift).

- The value of B is then added to restore the partial remainder in A to its previous value.
- The partial remainder is shifted to the left and the process is repeated again until all five quotient bits are formed.
- **Note** that while the partial remainder is shifted left, the quotient bits are shifted also and after five shifts, the quotient is in Q and the final remainder is in A .

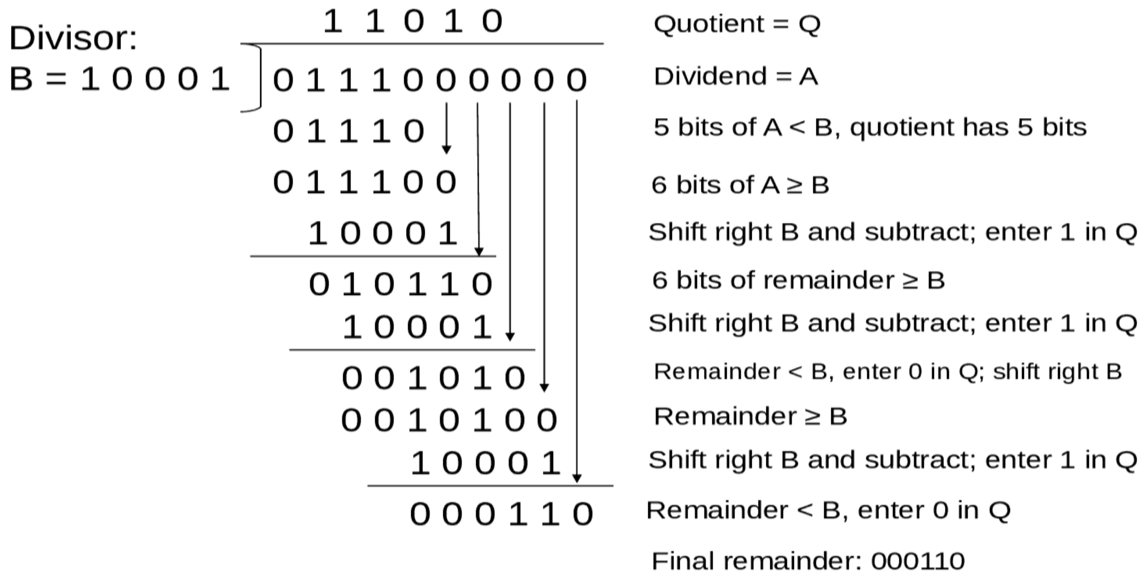
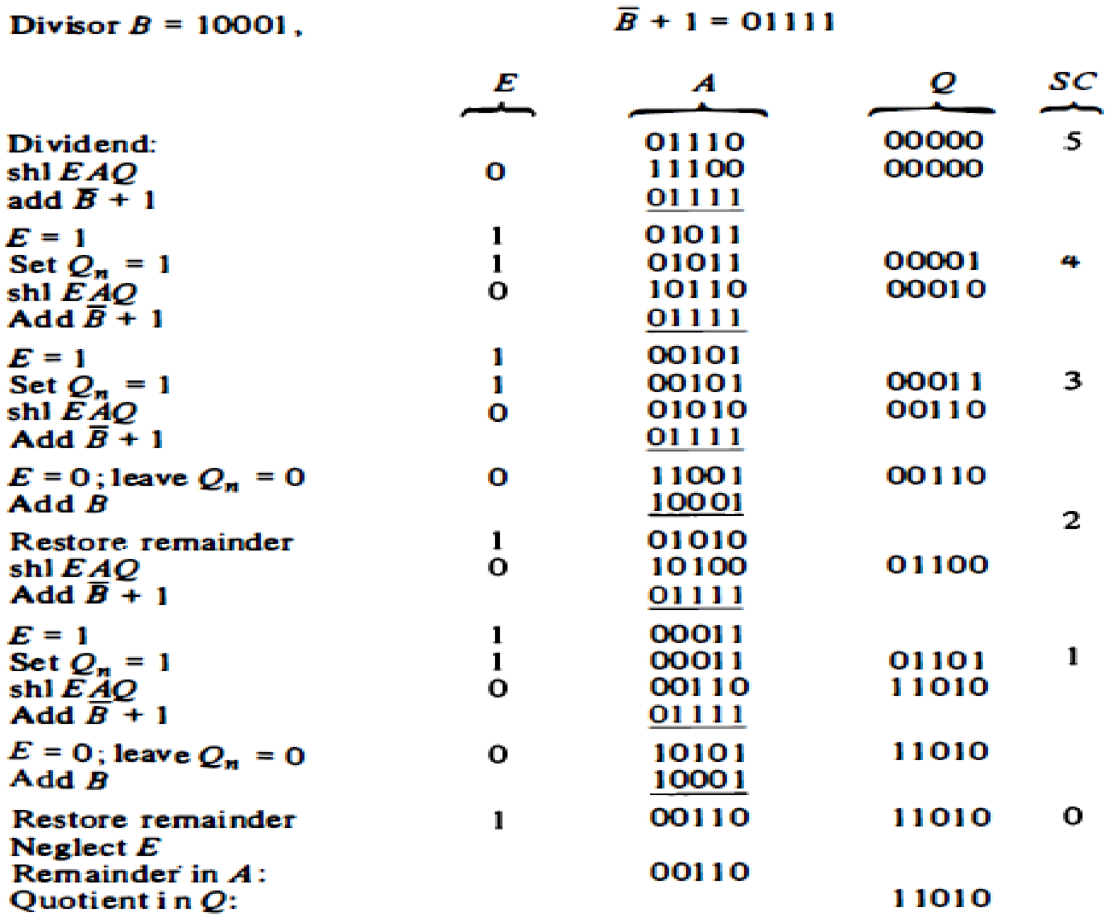


Figure 3-G: Example of binary division.



Divisor $B = 10001$, $\overline{B} + 1 = 01111$

	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\overline{B} + 1$		01111		
$E = 1$	1	01011		
Set $Q_n = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00101		
Set $Q_n = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	11001	00110	
Add B		10001		
Restore remainder	1	01010		2
shl EAQ	0	10100	01100	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00011		
Set $Q_n = 1$	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A:		00110		
Quotient in Q:			11010	

- Before showing the algorithm in flowchart form, we have to consider the sign of the result and a possible overflow condition.
- The sign of the quotient is determined from the signs of the dividend and the divisor.
- If the two signs are alike, the sign o f the quotient is plus. If they are unalike, the sign is minus. The sign of the remainder is the same as the sign of the dividend.

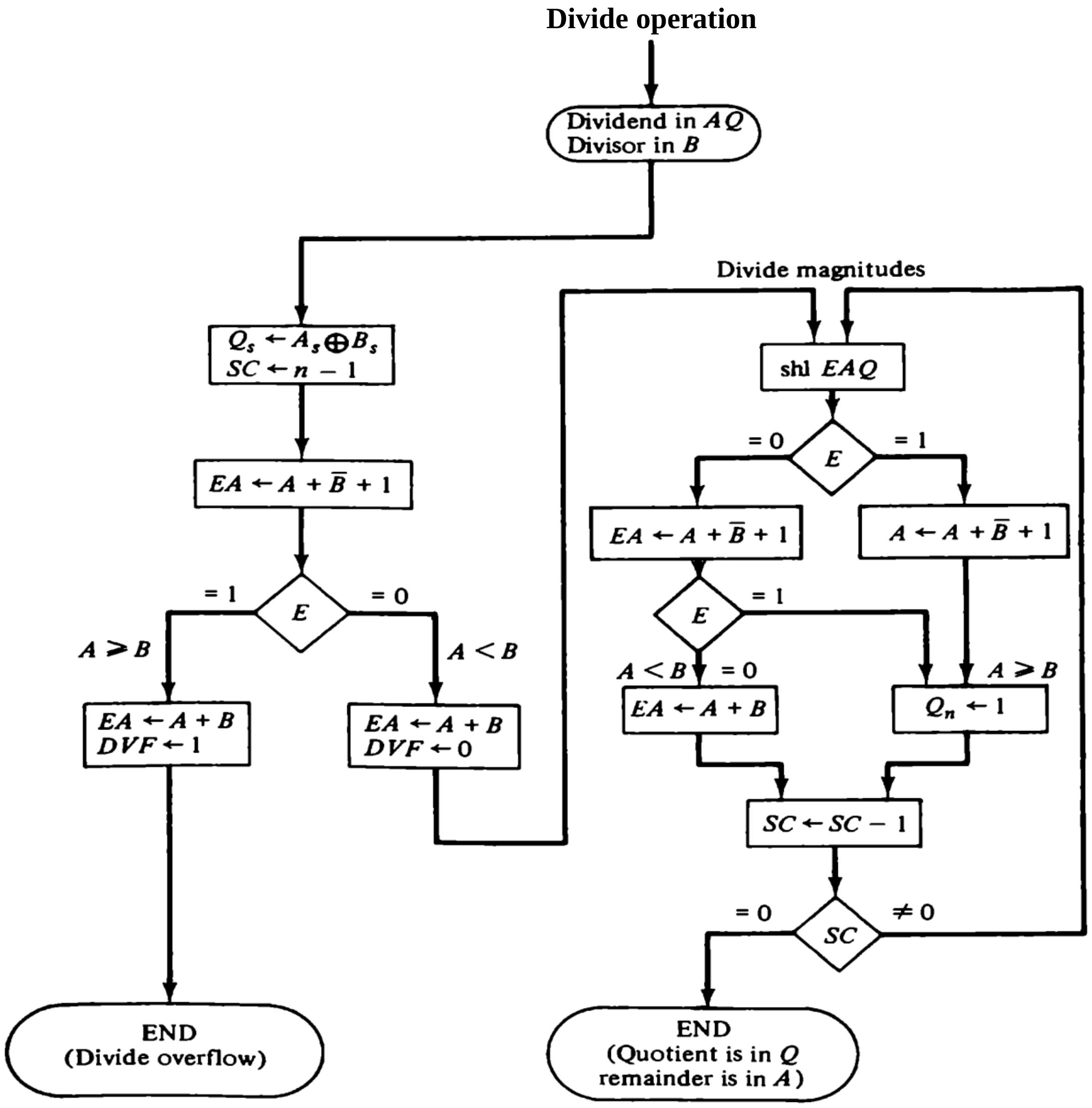
Divide Overflow:

- This divide-overflow condition must be avoided in normal computer operations because the entire quotient will be too long for transfer into a memory unit, that is, the same as the length of registers.
- When the dividend is twice as long as the divisor, the condition for overflow can be stated as follows:
    - A divide-overflow condition occurs if the high-order half bits of the dividend constitute a number greater than or equal to the divisor.
    - Another problem associated with division is the fact that a division by zero must be avoided.
- Overflow condition is usually detected when a special flip-flop is set. We will call it a divide-overflow flip-flop and label it DVF.

Hardware Algorithm:

The hardware divide algorithm is shown in the flowchart of Fig. 3-H .

- The dividend is in A and Q and the divisor in B . The sign of the result is transferred into Q, to be part of the quotient.
- A constant is set into the sequence counter SC to specify the number of bits in the quotient.
- A divide-overflow condition is tested by subtracting the divisor in B from half of the bits of the dividend stored in A.
- If A >=B, the divide-overflow flip-flop DVF is set and the operation is terminated prematurely.
- If A < B, no divide overflow occurs so the value of the dividend is restored by adding B to A.



Divide operation
Dividend in AQ Divisor in B
$Q_s \leftarrow A_s \oplus B_s$ $SC \leftarrow n-1$
$EA \leftarrow A+\overline{B}+1$
E
$=1$
$=0$
$A \geqslant B$
$A < B$
$EA \leftarrow A+B$ $DVF \leftarrow 1$
$EA \leftarrow A+B$ $DVF \leftarrow 0$
END (Divide overflow)
Divide magnitudes
shl EAQ
$=0$
$=1$
$EA \leftarrow A+\overline{B}+1$
$A \leftarrow A+\overline{B}+1$
$=1$
$A < B$
$=0$
$A \geqslant B$
$EA \leftarrow A+B$
$Q_n \leftarrow 1$
$SC \leftarrow SC-1$
SC
$=0$
$\neq 0$
END (Quotient is in Q remainder is in A)

Figure 3-H: Flowchart for divide operation

Decimal Arithmetic Unit:

- A CPU with an arithmetic logic unit can perform arithmetic micro operations with binary data. To perform arithmetic operations with decimal data, **it is necessary to convert the input decimal numbers to binary**, to perform all calculations with binary numbers, and to convert the results into decimal. This may be an efficient method in applications requiring a large number of calculations and a relatively smaller amount of input and output data.
- When the application calls for a large amount of input-output and a relatively smaller number of arithmetic calculations, it becomes convenient to do the internal arithmetic directly with the decimal numbers. **Computers capable of performing decimal arithmetic must store the decimal data in binary coded form**. The decimal numbers are then applied to a decimal arithmetic unit capable of executing decimal arithmetic micro operations.

BCD Addition:

- Consider the arithmetic addition of two decimal digits in BCD, together with a possible carry from a previous stage. Since each input digit does not exceed 9, the output sum cannot be greater than 9 + 9 + 1 = 19, the 1 in the sum being an input-carry.
- Suppose that we apply two BCD digits to a 4-bit binary adder. The adder will form the sum in binary and produce a result that may range from **0 to 19**.
- These binary numbers are listed in Table 3-I and are labeled by symbols K, Z_8, Z_4, Z_2, and Z_1.
- K is the carry and the subscripts under the letter Z represent the weights 8, 4, 2, and 1 that can be assigned to the four bits in the BCD code.
- The **first column** in the table lists the binary sums as they appear in the outputs of a 4-bit binary adder. The output sum of two decimal numbers must be represented in BCD and should appear in the form listed in the **second column** of the table.
- The problem is to find a simple rule by which the binary number in the first column can be converted to the correct BCD digit representation of the number in the second column.
- **In examining** the contents of the table, it is apparent that when the binary sum is equal to or less than 1001, the corresponding BCD number is identical and therefore no conversion is needed.
- **When the binary sum is greater than 1001**, we obtain a nonvalid BCD representation. The addition of **binary 6 (0110) to the binary sum converts it to the correct BCD representation** and also produces an output-carry as required.

TABLE 3-I: **Derivation of BCD Adder**

Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

→ One method of adding decimal numbers in BCD would be to employ one 4-bit binary adder and perform the arithmetic operation one digit at a time.

→ The low-order pair of BCD digits is first added to produce a binary sum. If the result is equal or greater than 1010, it is corrected by adding 0110 to the binary sum.

→ This second operation will automatically produce an output-carry for the next pair of significant digits. The next higher-order pair of digits, together with the input-carry, is then added to produce their binary sum. If this result is equal to or greater than 1010, it is corrected by adding 0110.

→ The procedure is repeated until all decimal digits are added.

BCD adder circuit:

❖ The logic circuit that detects the necessary correction can be derived from the table entries. It is obvious that a correction is needed when the binary sum has an output carry K = 1.

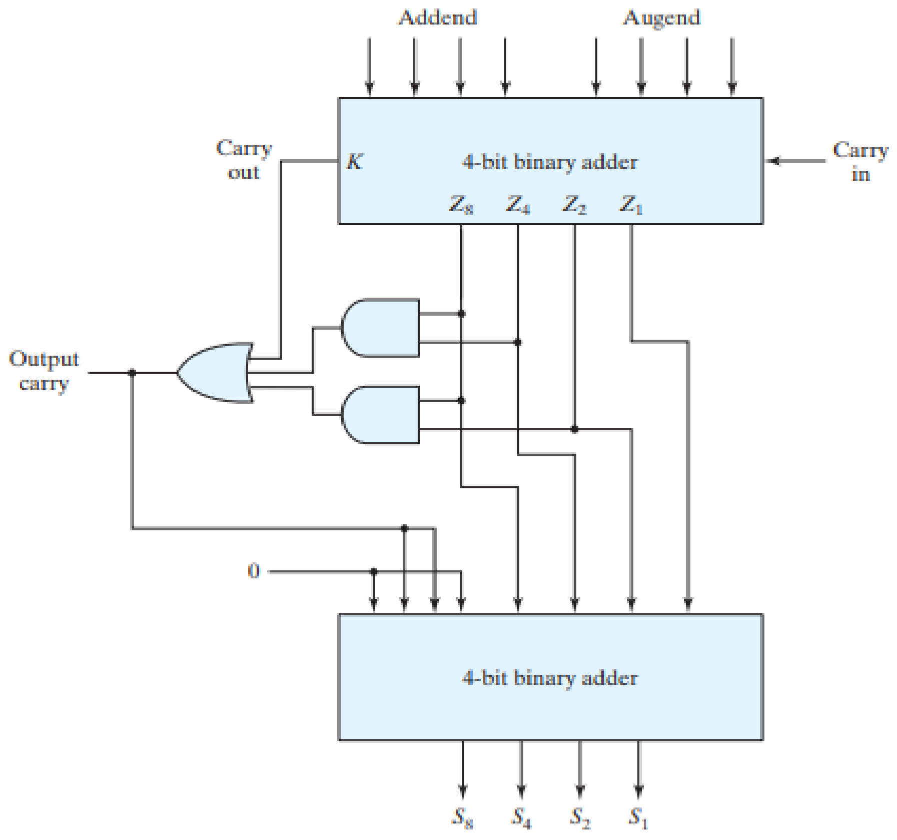
❖ The other six combinations from 1010 to 1 1 1 1 that need a correction have a 1 in position Z_8.To distinguish them from binary 1000 and 1001 which also have a 1 in position Z_8, we specify further that either Z_4 or Z_2, must have a 1.

❖ The condition for a correction and an output-carry can be expressed by the Boolean function

$$\mathbf{C = K + Z_8 Z_4 + Z_8 Z_2}$$

When C = 1, it is necessary to add **0110** to the binary sum and provide an output-carry for the next stage.

Figure 3-J: Block diagram of BCD adder



- ❖ A BCD adder is a circuit that adds two BCD digits in parallel and produces a sum digit also in BCD.
- ❖ A BCD adder must include the correction logic in its internal construction. To add 0110 to the binary sum, we use a second 4-bit binary adder as shown in Fig. 3-J.
- ❖ The two decimal digits, together with the input-carry, are first added in the top 4-bit binary adder to produce the binary sum. When the output-carry is equal to 0, nothing is added to the binary sum.
- ❖ When it is equal to 1, binary 0110 is added to the binary sum through the bottom 4-bit binary adder.
- ❖ The output-carry generated from the bottom binary adder may be ignored, since it supplies information already available in the output-carry terminal.

BCD Subtraction:

A straight subtraction of two decimal numbers will require a subtractor circuit that will be somewhat different from a BCD adder. It is more economical to perform the subtraction by taking the 9's or 10's complement of the subtrahend and adding it to the minuend. Since the BCD is not a self-complementing code, the 9' s complement cannot be obtained by complementing each bit in the code. It must be formed by a circuit that subtracts each BCD digit from 9.

- ❖ The 9's complement of a decimal digit represented in BCD may be obtained by complementing the bits in the coded representation of the digit provided a correction is included.
- ❖ **There are two possible correction methods**.
  - ✓ In the first method, binary 1010 (decimal 10) is added to each complemented digit and the carry discarded after each addition.
  - ✓ In the second method, binary 0110 (decimal 6) is added before the digit is complemented.

- ❖ **As a numerical illustration**, the 9's complement of BCD 0111 (decimal 7) is computed by first complementing each bit to obtain 1000. Adding binary 1010 and discarding the carry, we obtain 0010 (decimal 2). By the second method, we add 0110 to 0111 to obtain 1101. Complementing each bit, we obtain the required result of 0010.

BCD Subtraction Circuit:

- ❖ The 9's complement of a BCD digit can also be obtained through a combinational circuit. When this circuit is attached to a BCD adder, the result is a BCD adder/subtractor.
- ❖ Let the subtrahend (or addend) digit be denoted by the four binary variables B_8, B_4, B_2, and B_1.
- ❖ Let M be a mode bit that controls the add/subtract operation.
  - ✓ When M = 0, the two digits are added;
  - ✓ When M = 1, the digits are subtracted.
- ❖ Let the binary variables x_8, x_4, x_2, and x_1 be the outputs of the 9's complementer circuit.
- ❖ By an examination of the truth table for the circuit, it may be observed that-
  - ✓ B_1 should always be complemented;
  - ✓ B_2 is always the same in the 9' s complement as in the original digit;
  - ✓ x4 is 1 when the exclusive-OR of B_2 and B_4 is 1; and
  - ✓ x_8 is 1 when B_8 B_4 B_2 = 000.
- ❖ The Boolean functions for the 9' s complementer circuit are From these equations we see that x = B when M = 0. When M = 1, the x outputs produce the 9' s complement of B

$$x_1 = B_1M' + B_1'M$$

$$x_2 = B_2$$

$$x_4 = B_4M' + (B_4'B_2 + B_4B_2')M$$

$$x_8 = B_8M' + B_8'B_4'B_2'M$$

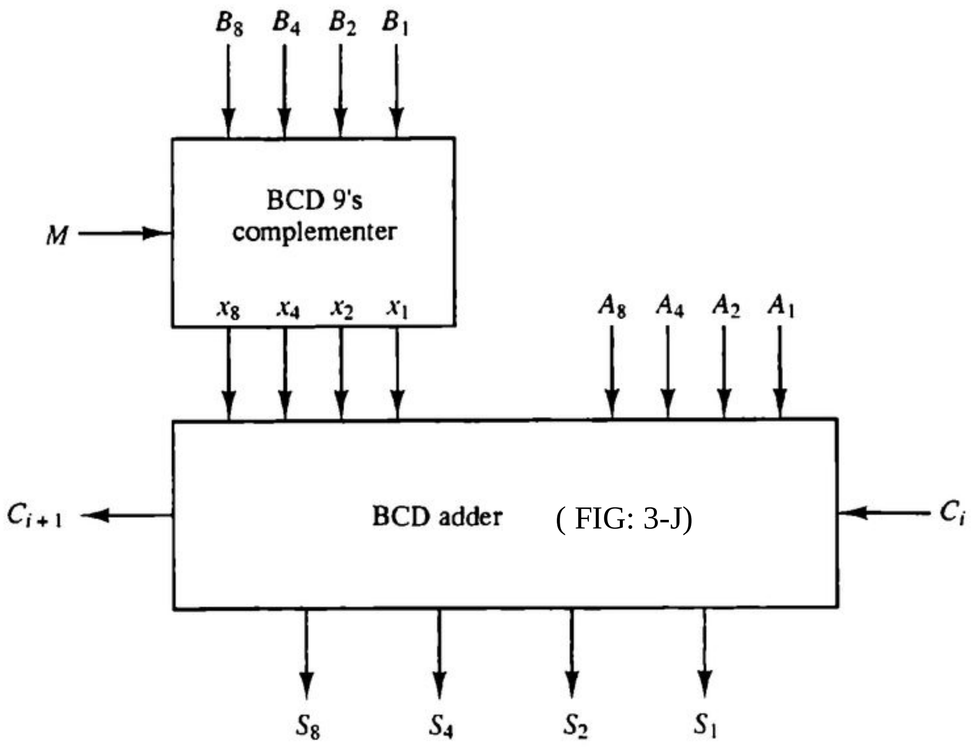


Figure :3-K: One stage of a decimal arithmetic unit.

- One stage o f a decimal arithmetic unit that can add o r subtract two BCD digits is shown in Fig.3-K. It consists of a BCD adder and a 9's complementer. The mode M controls the operation of the unit.
  - ✓ With M = 0, the S outputs form the sum of A and B.
  - ✓ With M = 1, the S outputs form the sum of A plus the9's complement of B.
- For numbers with n decimal digits we need n such stages. The output carry C_{i+1} from one stage must be connected to the input carry C_i of the next-higher-order stage.
- The best way to subtract the two decimal numbers is to let M = 1 and apply a 1 to the input carry C_1 of the first stage. **The outputs will form the sum of A plus the 10's complement of B, which is equivalent to a subtraction operation if the carry-out of the last stage is discarded**.

Example:

Fig. 3.K shows the logic diagram of the circuit to implement above mentioned steps to perform BCD subtraction using 9’s complement method. As shown in the Fig. 3.K, first binary adder finds the 9’s complement of the negative number. It does this by inverting each bit of BCD number and adding 10 $(1\ 0\ 1\ 0)_2$ to it. Let us find the 9’s complement of 2

$$\begin{array}{rrl} & 0\,0\,1\,0 & \leftarrow \text{BCD for 2} \\ & 1\,1\,0\,1 & \leftarrow \text{Inverting each bit} \\ & +\ 1\,0\,1\,0 & \leftarrow \text{Add 10 } (1010_2) \\ \hline \text{Ignore carry} \rightarrow & 1\,0\,1\,1\,1 & \leftarrow \text{9's complement for 2} \end{array}$$

Next two 4-bit binary adders perform the BCD addition. The last adder finds the 9’s complement of the result if carry is not generated after BCD addition otherwise it adds carry in the result. (See Fig. 3.34 on previous page).

From the above examples we can summarize steps for 10’s complement BCD subtraction as follows.

- Find the 10’s complement of a negative number
- Add two numbers using BCD addition
- If carry is not generated find the 10’s complement of the result.

Fig. 3.35 shows the logic diagram of the circuit to implement above mentioned steps to perform BCD subtraction using 10’s complement method. As shown in the Fig. 3.35, first binary adder finds the 10’s complement of the negative number (9’s complement + 1). Next two 4-bit binary adders perform the BCD addition. Finally, last 4-bit binary.