(23CS304) COMPUTER ORGANIZATION & ARCHITECTURE
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 REPRESENTATION

Data types: Binary information in digital computers is stored in memory or processor registers. Registers contain either data or control information. Control information is a bit or a group of bits used to specify the sequence of command signals needed for manipulation of the data in other registers. Data are numbers and other binary-coded information that are operated on, to achieve required computational results.

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.
Given below is different **basic computer data types** that computer uses:

- **Numeric data** – Integer and Real numbers
- **Non-numeric data** – Character data, address data, logical data

Numeric data types

It can be of the following two types:

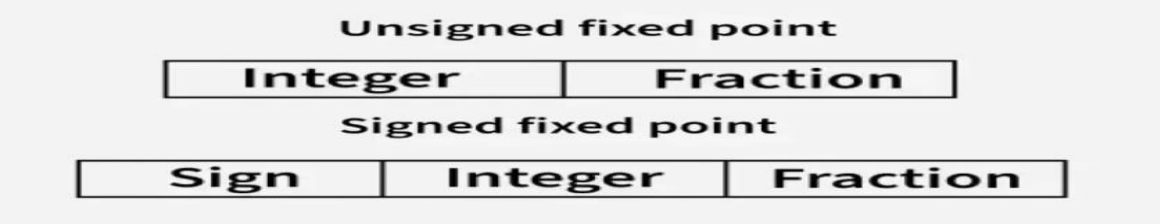
- Integers
- Real Numbers

Real numbers can be represented as:

1. Fixed point representation
2. Floating point representation

Fixed-point representation

It is a method of storing and computing fractional numbers by fixing the position of the binary (or radix) point. Unlike the flexible scaling of floating-point numbers, fixed-point numbers have a predetermined and static number of bits allocated for the integer part and the fractional part.



Unsigned Representation

For example, signifies an 8-bit fixed-point number, the rightmost 3 bits of which are fractional.

00010.110

$= 1 * 2^1 + 1 * 2^{-1} + 1 * 2^{-2}$

$= 2 + 0.5 + 0.25$

$= 2.75$

Signed Representation

Signed binary numbers (+ve or -ve) can be represented in one of three ways:

1. Sign-Magnitude form

   In sign-magnitude form, the number's sign is represented by the MSB (Most Significant Bit also called as Leftmost Bit), while its magnitude is shown by the remaining bits (In the case of 8-bit representation Leftmost bit is the sign bit and remaining bits are magnitude bit).

   $55_{10} = 00110111_2$
   $-55_{10} = 10110111_2$

2. 1's complement form

   $55_{10} = 00110111_2$
   $-55_{10} = 10110111_2$

3. 2's complement form

   $55_{10} = 11001000 + 1$ (1's complement + 1 = 2's complement)
   $-55_{10} = 11001001_2$

16-bit Fixed-Point Number: (Sign Bit: 1 bit, Integer Part: 5 bits, Fractional Part: 10 bits)

- **Sign Bit:** The most significant bit (MSB) indicates the sign (0 for positive, 1 for negative). Negative numbers are typically represented using two's complement notation.

- **Integer Part:** The bits to the left of the binary point represent the integer part of the number.
- **Fractional Part:** The bits to the right of the binary point represent the fractional part.

Example: Representing 5.75 using an 8-bit Q4.3 format

An 8-bit Q4.3 format means 4 integer bits and 3 fractional bits (plus a sign bit for signed numbers). Let's use two's complement for signed numbers.

1. Split the number:

- Integer part: 5
- Fractional part: 0.75

2. Convert the integer part to binary:

- $5_{10} = 101_2$
- The integer part needs to be padded to fit the 4-bit allocation: `0101`.

3. Convert the fractional part to binary:

- To convert the fractional part (0.75) to binary, repeatedly multiply the fraction by 2 and take the integer part.
  - $0.75 \times 2 = 1.50 \rightarrow$ bit is **1**
  - $0.50 \times 2 = 1.00 \rightarrow$ bit is **1**
  - $0.00 \times 2 = 0.00 \rightarrow$ bit is **0**
- The fractional part is 0.110_2.

4. Combine the parts:

- The binary fixed-point number is 0101.110_2.
- Since the number is positive, the sign bit is `0`.
- The full 8-bit representation is `01011100`. The binary point is implicit, placed between the 4th and 5th bit.

5. Verification:

To confirm, let's decode the binary representation:

- Sign: 0 (positive)
- Integer part: $0101_2 = 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 4 + 1 = 5$
- Fractional part: $110_2 = 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} = 0.5 + 0.25 = 0.75$
- Result: $5 + 0.75 = 5.75$

Example: Representing -2.5 using 8-bit Q4.3 format

1. Represent the magnitude:

- First, represent the positive value 2.5.
- Integer part: $2_{10} = 0010_2$ (4 bits).
- Fractional part: $0.5_{10} = 0.1_2$ (padded to 3 bits: `100`).
- The binary representation of 2.5 is `0010.100`.

2. Apply two's complement:

- The total 8-bit pattern for +2.5 is `00101000`.
- To get the two's complement for -2.5:
  - Invert all bits: `11010111`
  - Add 1: `11011000`
- The 8-bit Q4.3 fixed-point representation for -2.5 is `11011000`.

Floating-point representation:

It is a method used by computers to store and approximate real numbers, which include both integers and fractional values. This system, standardized by the IEEE 754 format, uses a binary version of scientific notation and can represent a vast range of values from very small to very large.

Floating-point numbers are stored by splitting them into three distinct parts:

- **Sign (S):** A single bit that indicates if the number is positive (0) or negative (1).
- **Exponent (E):** A field that represents the power of the base (2 in binary) to scale the number.
- **Mantissa (M) or Significand:** The field that holds the significant digits of the number, determining its precision.

The value of the number is calculated with the formula: $S \times M \times B^E$
The most common implementation is the IEEE 754 **single-precision** (32-bit) format.
Sign bit: 1 bit

- **Exponent field:** 8 bits
- **Mantissa field:** 23 bits
- B- base

The most common implementation is the IEEE 754 **double-precision** (64-bit) format.
Sign bit: 1 bit

- **Exponent field:** 11 bits
- **Mantissa field:** 52 bits
- B- base

Exponent biasing and normalization

To accommodate both positive and negative exponents without using an extra sign bit, a fixed **bias** is added to the exponent.
For the single-precision format, the bias is $(1.N)2^{E-127}$.
For the double-precision format, the bias is $(1.N)2^{E-1023}$.

Additionally, to save space and increase precision, a process called **normalization** is used for most numbers. All normalized binary floating-point numbers are shifted so they have a leading '1' before the binary point. Since this '1' is always present, it does not need to be stored and is known as the **hidden bit** or **implicit bit**.

Example: Representing the decimal number 5.75

Here is a step-by-step process for converting 5.75 into its 32-bit floating-point representation:

Step 1: Determine the sign

- The number 5.75 is positive, so the sign bit is `0`.

Step 2: Convert to binary and normalize

- Convert the integer and fractional parts to binary:
  - $5_{10} = 101_2$
  - $0.75_{10} = 0.11_2$ (as $0.5 + 0.25$)
- Combine them: $5.75_{10} = 101.11_2$
- Normalize the binary number by shifting the decimal point until there is a single '1' before it.
  - $101.11_2 = 1.0111 \times 2^2$
- The mantissa is the fractional part: `0111`.

Step 3: Calculate the biased exponent

- The actual exponent is 2.
- Add the bias for single-precision (127):
  - Biased exponent = $2 + 127 = 129$
- Convert the biased exponent to an 8-bit binary number:
  - $129_{10} = 10000001_2$

Step 4: Assemble the 32-bit floating-point number

- Combine the three components (sign, exponent, and mantissa):
  - **Sign:** `0`
  - **Exponent:** `10000001`
  - **Mantissa:** `0111` (padded with zeros to fill the remaining 23 bits) `01110000000000000000000`
- The full 32-bit binary representation is: `0 10000001 01110000000000000000000`

Example: Representing the decimal number -0.75

This example follows the same steps for a negative number:

Step 1: Determine the sign

- The number -0.75 is negative, so the sign bit is `1`.

Step 2: Convert to binary and normalize

- The absolute value is $0.75_{10} = 0.11_2$.
- Normalize the binary number:
  - $0.11_2 = 1.1 \times 2^{-1}$
- The mantissa is the fractional part: `1`.

Step 3: Calculate the biased exponent

- The actual exponent is -1.

- Add the bias (127):
  - Biased exponent = $-1 + 127 = 126$
- Convert the biased exponent to an 8-bit binary number:
  - $126_{10} = 01111110_2$

Step 4: Assemble the 32-bit floating-point number

- **Sign:** `1`
- **Exponent:** `01111110`
- **Mantissa:** `1` (padded with zeros to fill the remaining 23 bits) `10000000000000000000000`
- The full 32-bit binary representation is: `1 01111110 10000000000000000000000`

TABLE 3-6 Four Different Binary Codes for the Decimal Digit

Decimal digit	BCD 8421	2421	Excess-3	Excess-3 gray
0	0000	0000	0011	0010
1	0001	0001	0100	0110
2	0010	0010	0101	0111
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1100
6	0110	1100	1001	1101
7	0111	1101	1010	1111
8	1000	1110	1011	1110
9	1001	1111	1100	1010
	1010	0101	0000	0000
Unused	1011	0110	0001	0001
bit	1100	0111	0010	0011
combi-	1101	1000	1101	1000
nations	1110	1001	1110	1001
	1111	1010	1111	1011

- Binary codes for decimal digits require a minimum of four bits
- Other codes besides BCD exist to represent decimal digits

- The 2421 code and the excess-3 code are both *self-complementing*
- The 9's complement of each digit is obtained by complementing each bit in the code
- The 2421 code is a *weighted code*
- The bits are multiplied by indicated weights and the sum gives the decimal digit
- The excess-3 code is obtained from the corresponding BCD code added to 3

Error Detection Codes

- Transmitted binary information is subject to noise that could change bits 1 to 0 and vice versa
- An *error detection code* is a binary code that detects digital errors during transmission
- The detected errors cannot be corrected, but can prompt the data to be retransmitted
- The most common error detection code used is the *parity bit*

TABLE 3-5 4-Bit Gray Code

Binary code	Decimal equivalent	Binary code	Decimal equivalent
0000	0	1100	8
0001	1	1101	9
0011	2	1111	10
0010	3	1110	11
0110	4	1010	12
0111	5	1011	13
0101	6	1001	14
0100	7	1000	15

A parity bit is an extra bit included with a binary message to make the total number of 1's either odd or even

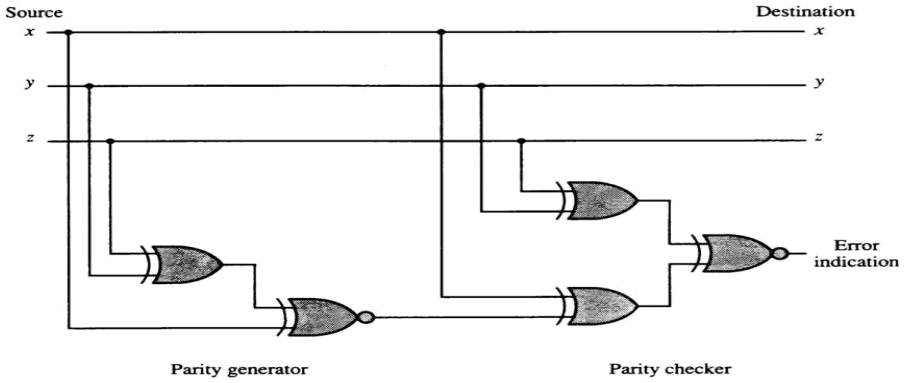
TABLE 3-7 Parity Bit Generation

Message *xyz*	*P*(odd)	*P*(even)
000	1	0
001	0	1
010	0	1
011	1	0
100	0	1
101	1	0
110	1	0
111	0	1

- The P(odd) bit is chosen to make the sum of 1's in all four bits odd
- The even-parity scheme has the disadvantage of having a bit combination of all 0's
- Procedure during transmission:
  - At the sending end, the message is applied to a *parity generator*
  - The message, including the parity bit, is transmitted
  - At the receiving end, all the incoming bits are applied to a *parity checker*
  - Any odd number of errors are detected

- Parity generators and checkers are constructed with XOR gates (odd function)
- An odd function generates 1 iff an odd number if input variables are 1

Figure 3-3 Error detection with odd parity bit.



COMPUTER ARITHMETIC

Introduction:

Data is manipulated by using the arithmetic instructions in digital computers. Data is manipulated to produce results necessary to give solution for the computation problems. The Addition, subtraction, multiplication and division are the four basic arithmetic operations. If we want then we can derive other operations by using these four operations.

To execute arithmetic operations there is a separate section called arithmetic processing unit in central processing unit. The arithmetic instructions are performed generally on binary or decimal data. Fixed-point numbers are used to represent integers or fractions. We can have signed or unsigned negative numbers. Fixed-point addition is the simplest arithmetic operation.

If we want to solve a problem then we use a sequence of well-defined steps. These stepss are collectively called algorithm. To solve various problems we give algorithms.

In order to solve the computational problems, arithmetic instructions are used in digital computers that manipulate data. These instructions perform arithmetic calculations.

And these instructions perform a great activity in processing data in a digital computer. As we already stated that with the four basic arithmetic operations addition, subtraction, multiplication and division, it is possible to derive other arithmetic operations and solve scientific problems by means of numerical analysis methods.

A processor has an arithmetic processor(as a sub part of it) that executes arithmetic operations. The data type, assumed to reside in processor, registers during the execution of an arithmetic instruction. Negative numbers may be in a signed magnitude or signed complement representation. There are three ways of representing negative fixed point - binary numbers signed magnitude, signed 1's complement or signed 2's complement. Most computers use the signed magnitude representation for the mantissa.

Addition and Subtraction :(Addition and Subtraction with Signed –Magnitude Data)

We designate the magnitude of the two numbers by A and B. Where 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 4.1. 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 present a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0.

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

Addition and Subtraction of Signed-Magnitude Numbers:

Eight Conditions for Signed-Magnitude Addition/Subtraction

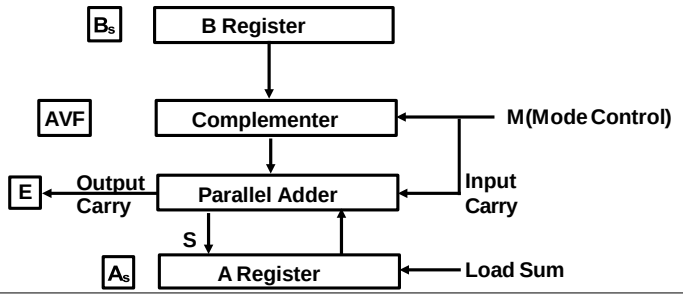
Operation	ADD Magnitudes	SUBTRACT Magnitudes		
		A > B	A < B	A = B
(+A) + (+B)	+ (A + B)			
(+A) + (-B)		+ (A – B)	- (B – A)	+ (A – B)
(-A) + (+B)		- (A – B)	+ (B – A)	+ (A – B)
(-A) + (-B)	- (A + B)			
(+A) - (+B)		+ (A – B)	- (B – A)	+ (A – B)
(+A) - (-B)	+ (A + B)			
(-A) - (+B)	- (A + B)			
(-A) - (-B)		- (A – B)	+ (B – A)	+ (A – B)

Hardware Implementation:

SIGNED MAGNITUDEADDITION AND SUBTRACTION

Addition: A + B ; A: Augend; B: Addend
Subtraction: A - B: A: Minuend; B: Subtrahend

Hardware Implementation



Flowchart:

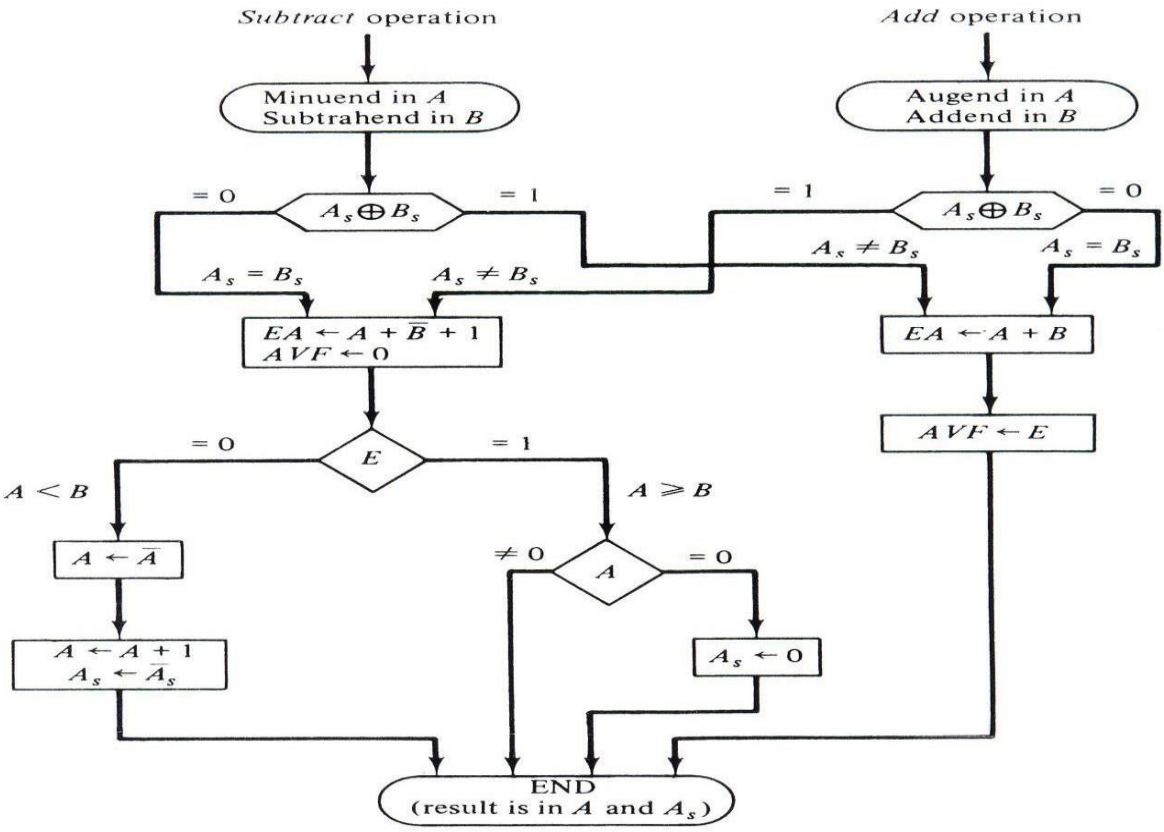


Figure 10-2 Flowchart for add and subtract operations.

Algorithm:

- The flowchart is shown in above Figure. The two signs A, and B, 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.

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

- The magnitudes are added with a microoperation EA 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 subtracted 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 complemented B. No overflow can occur if the numbers are subtracted so AVF is cleared to 0.

- 1 in E indicates that A >= B and the number in A is the correct result. If this numbs is zero, the sign A must be made positive to avoid a negative zero.

- 0 in E indicates that A < B. For this case it is necessary to take the 2's complement of the value in A. The operation can be done with one microoperation A A' +1.

- However, we assume that the A register has circuits for microoperations complement and increment, so the 2's complement is obtained from these two microoperations.

- In other paths of the flowchart, the sign of the result is the same as the sign of A. so no change in A is required. However, when A < B, the sign of the result is the complement of the original sign of A. It is then necessary to complement A, to obtain the correct sign.

- The final result is found in register A and its sign in As. The value in AVF provides an overflow indication. The final value of E is immaterial.

- Figure 7.2 shows a block diagram of the hardware for implementing the addition and subtraction operations.

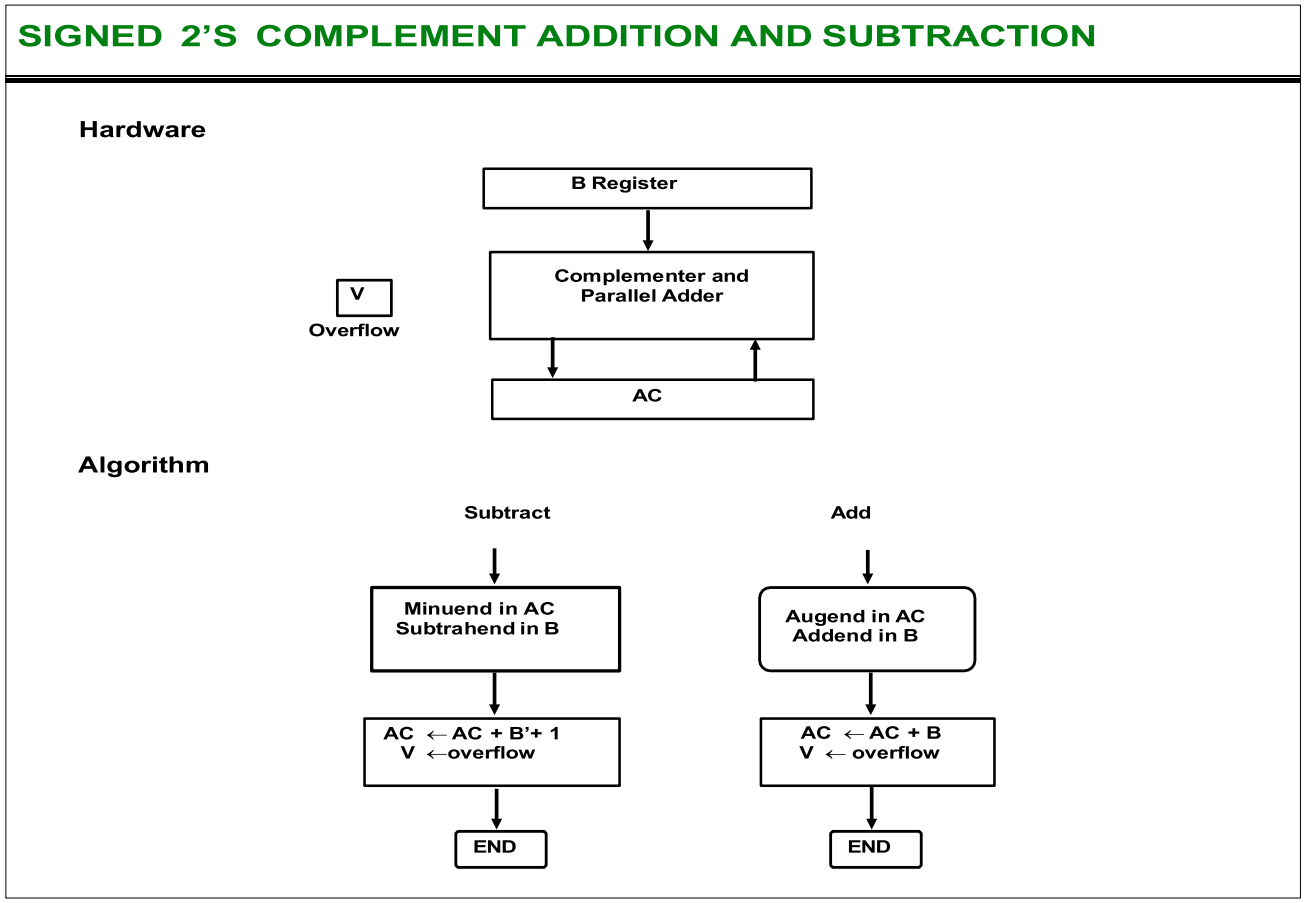
  It consists of registers A and B and sign flip-flops As and Bs. Subtraction is done by adding A to the 2's complement of B.

- The output carry is transferred to flip-flop E , where it can be checked to determine the relative magnitudes of two numbers.

  - The add-overflow flip-flop *AVF* holds the overflow bit when A and B are added.

- The A register provides other microoperations that may be needed when we specify the sequence of steps in the algorithm.

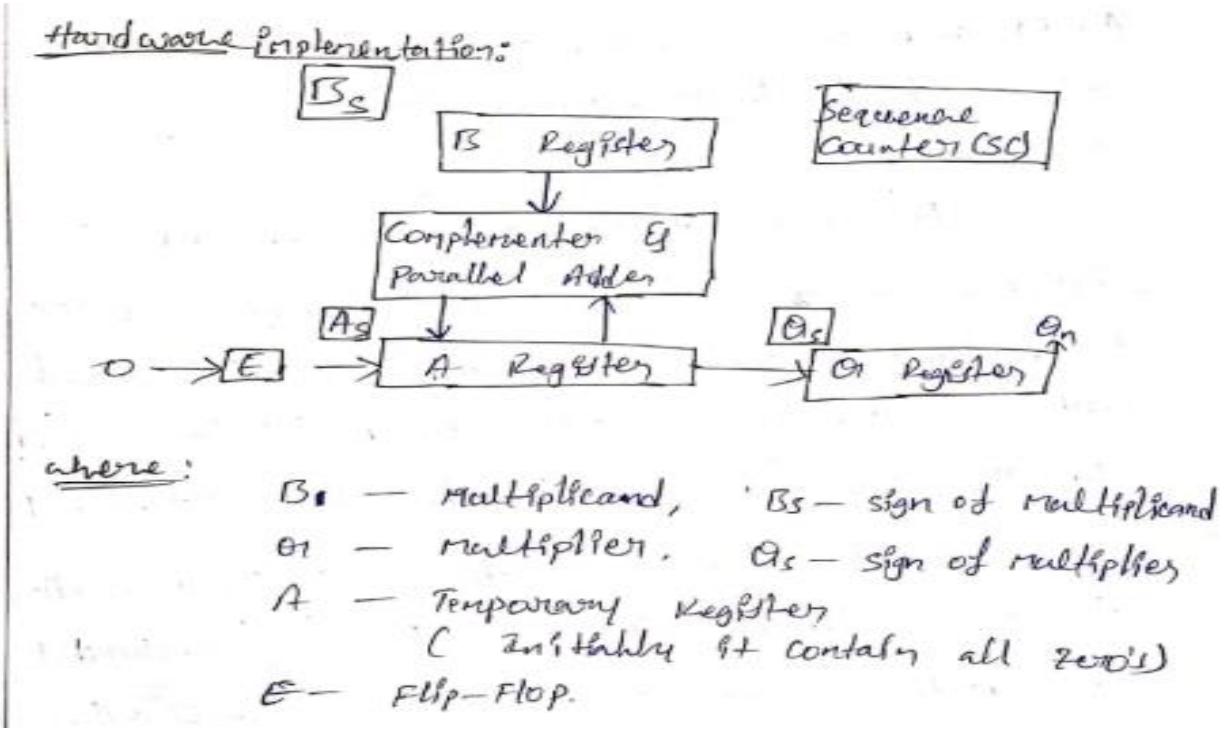
Addition and Subtraction with Signed 2's Complement–Magnitude Data



Multiplication Algorithm: (Multiplication with signed magnitude)

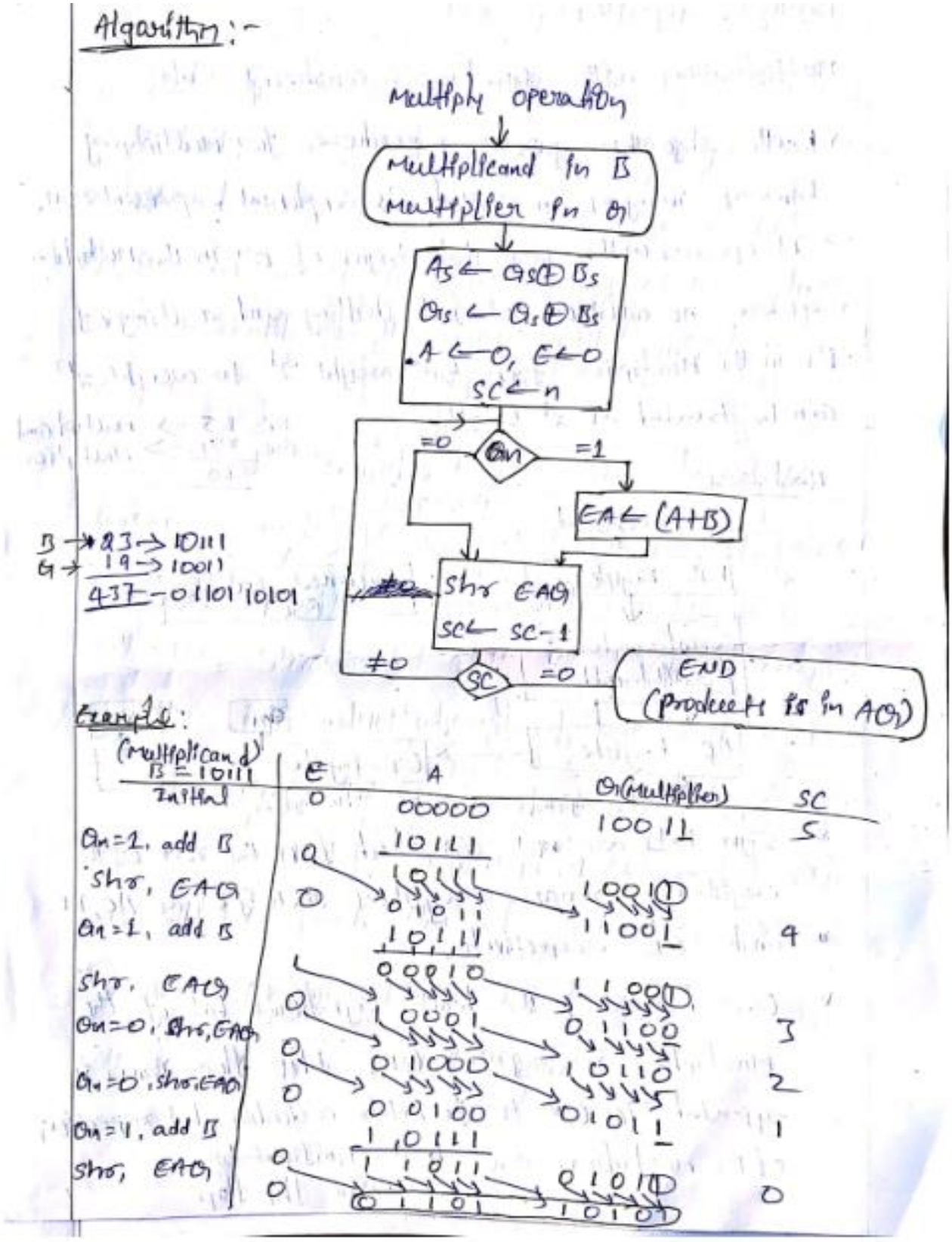
In the beginning, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in Bs and Qs respectively. We compare the signs of both A and Q and set to corresponding sign of the product since a double-length product will be stored in registers A and Q. Registers A and E are cleared and the sequence counter SC is set to the number of bits of the multiplier. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of n-1 bits.
Now, the low order bit of the multiplier in Qn is tested. If it is 1, the multiplicand (B) is added to present partial product (A), 0 otherwise. 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. When SC = 0 we stops the process.

Hardware Implementation:



Qn - least significant bit

Flowchart and Example:



Booth's algorithm (Multiplication with signed 2's Complement)

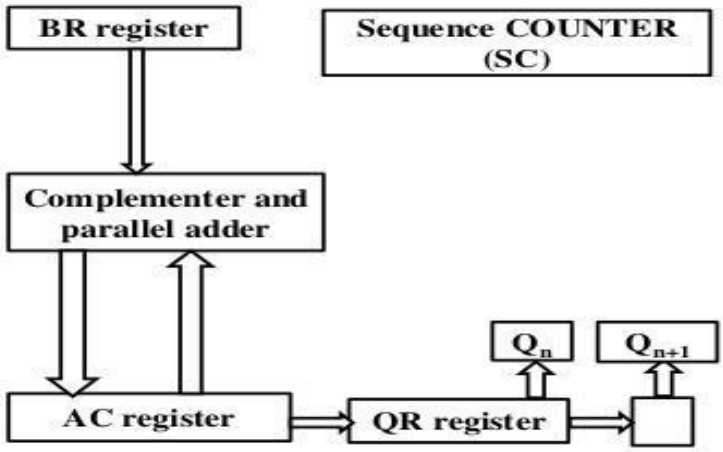
- Booth algorithm gives a procedure for multiplying binary integers in signed- 2's complement representation.
- It operates on the fact that strings of 0's in the multiplier require no addition but just shifting, and a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} - 2^m$.

- For example, the binary number 001110 (+14) has a string 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.

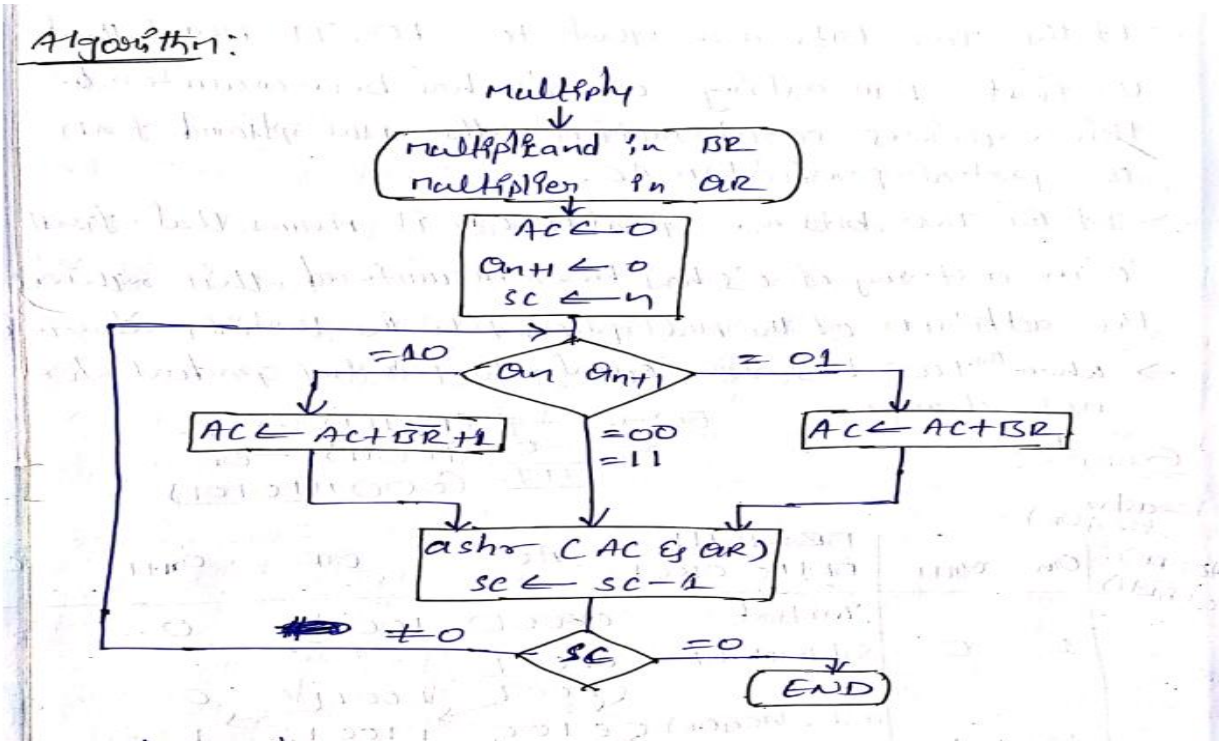
- Thus the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.

Hardware for Booth Algorithm

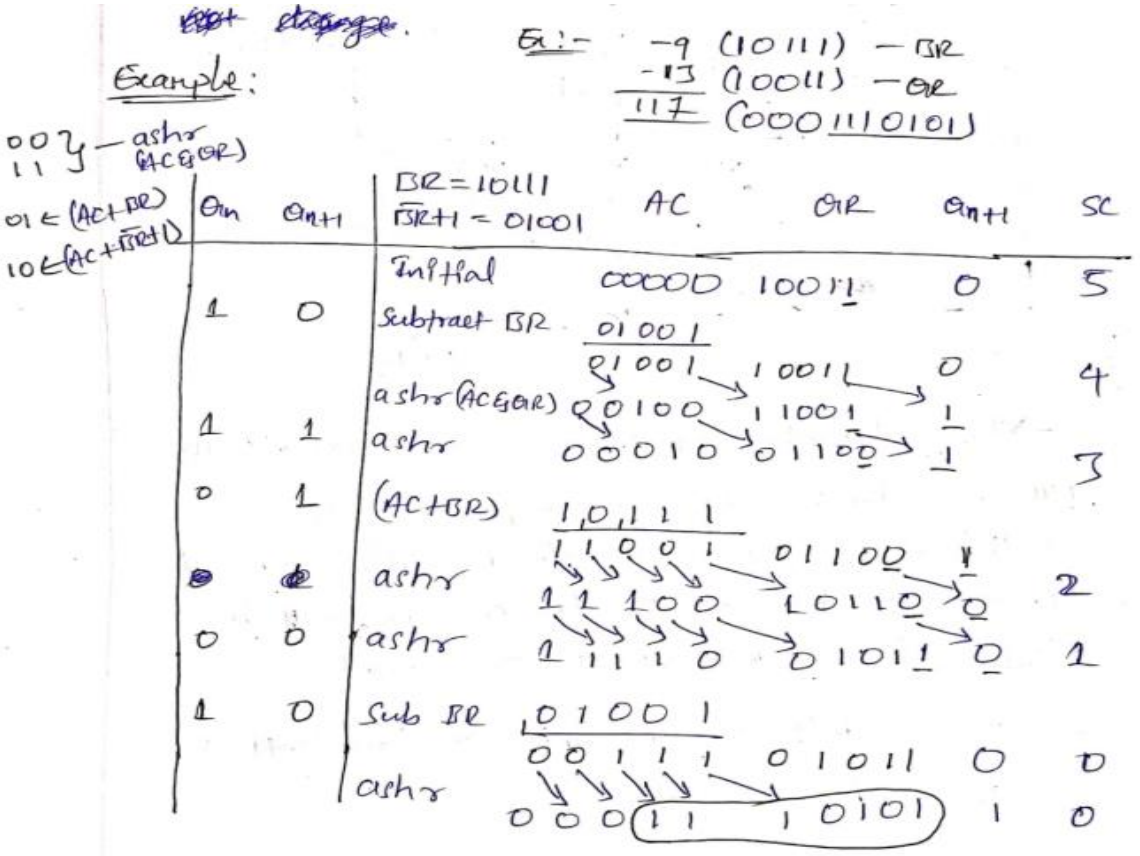
- Sign bits are not separated from the rest of the registers
- rename registers A,B, and Q as AC,BR and QR respectively
- Q_n designates the least significant bit of the multiplier in register QR
- Flip-flop Qn+1 is appended to QR to facilitate a double bit inspection of the multiplier



Flowchart for Booth's Algorithm:



Example:



Algorithm:

- As in all multiplication schemes, booth algorithm requires examination of the multiplier bits and shifting of partial product.
- Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial, 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 in a string of 0's in the multiplier.

3.The partial product does not change when multiplier bit is identical to the previous multiplier bit.

- The algorithm works for positive or negative multipliers in 2's complement representation.
- This is because a negative multiplier ends with a string of 1's and the last operation will be a subtraction of the appropriate weight.
- The two bits of the multiplier in Qn and Qn+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.

Division Algorithms(Division with Signed-Magnitude Data):

Division of two fixed-point binary numbers in signed magnitude representation is performed with paper and pencil by a process of successive compare, shift and subtract operations. Binary division is much simpler than decimal division because here the quotient digits are either 0 or 1 and there is no need to estimate how many times the dividend or partial remainder fits into the divisor. The division process is described in Figure

```
                  00001010 1   Quotient
Divisor  1101 ) 10001001 0     Dividend
                -1101
                 10000
                -1101
                  1110
                 -1101
                     1         Remainder
```

The devisor is compared with the five most significant bits of the dividend. Since the 5-bit number is smaller than B, we again repeat the same process. Now the 6-bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend. Now we shift the divisor once to the

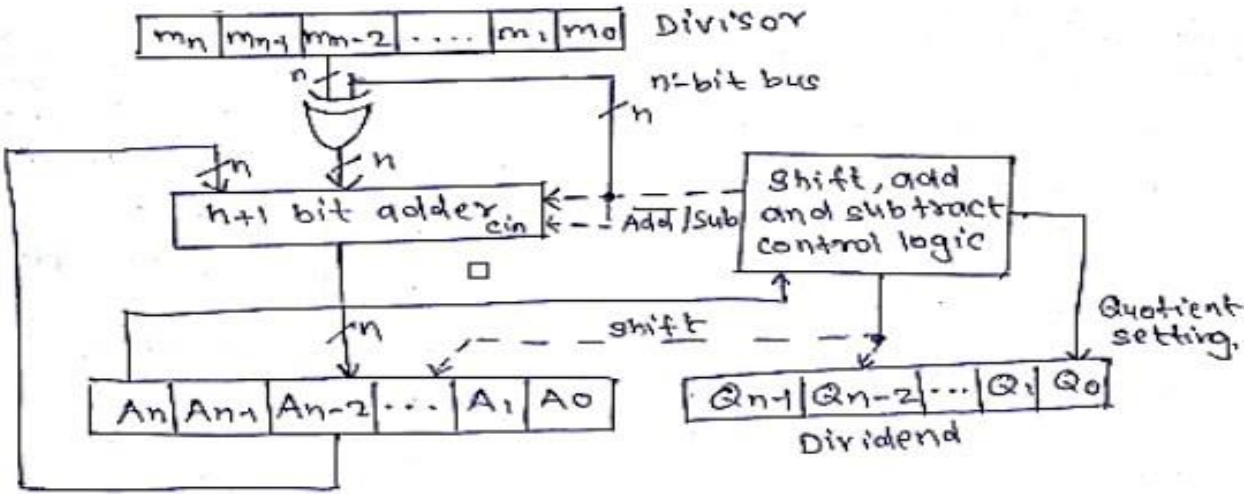
right and subtract it from the dividend. The difference is known as a partial remainder because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder. Comparing a partial remainder with the divisor continues the process. If the partial remainder 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. Obviously the result gives both a quotient and a remainder.

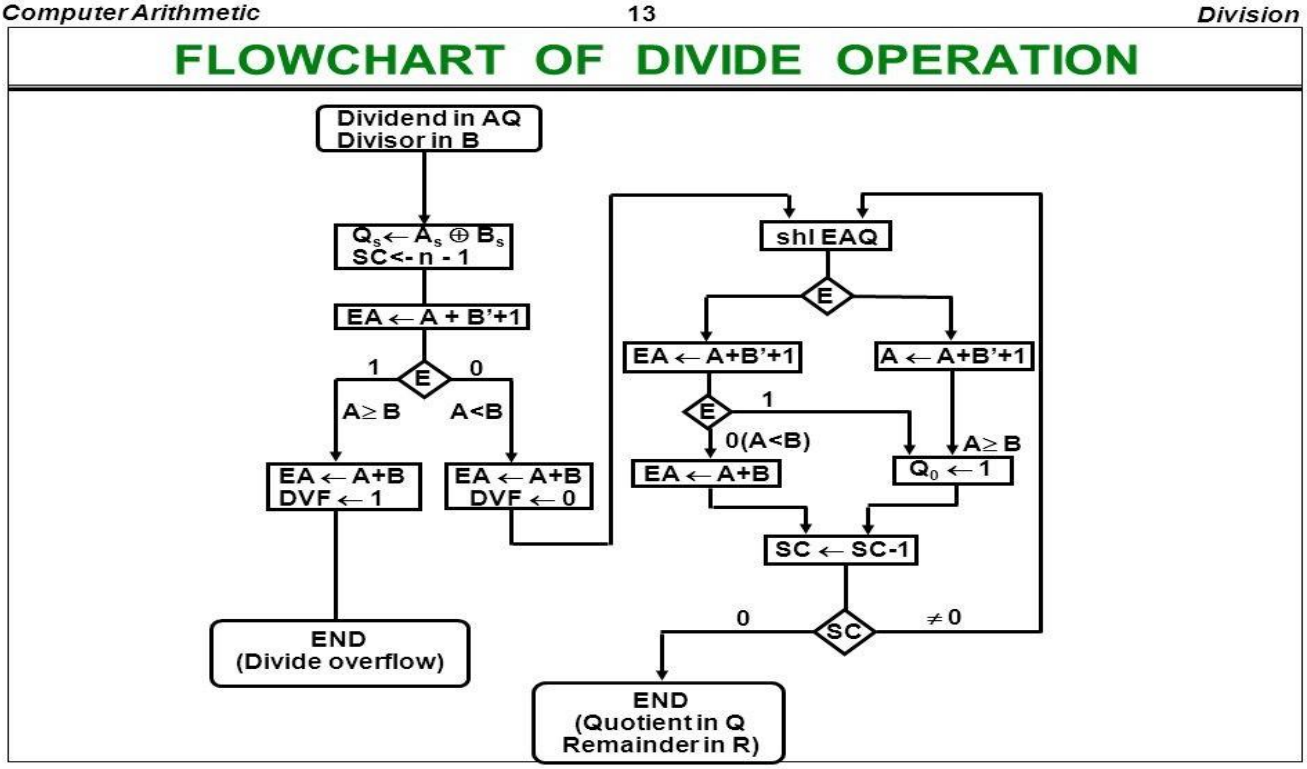
Hardware Implementation for Signed-Magnitude Data:

In hardware implementation for signed-magnitude data in a digital computer, it is convenient to change the process slightly. Instead of shifting the divisor to the right, two dividends, or partial remainders, are shifted to the left, thus leaving the two numbers in the required relative position. Subtraction is achieved by adding A to the 2's complement of B. End carry gives the information about the relative magnitudes.

The hardware required is identical to that of multiplication. Register EAQ is now shifted to the left with 0 inserted into Qn and the previous value of E is lost. The example is given in Figure 4.10 to clear the proposed division process. 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.



Hardware Implementation for Signed-Magnitude Data



Example of Binary Division with Digital Hardware:
=>Where: Divisor: 17(B=10001), Dividend: 448(01110 00000), Quotient:26(11010), Remainder: 6(00110).

Divisor B = 10001	*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		2
Restore remainder	1	01010		
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	

Floating-point Arithmetic operations:

In many high-level programming languages, we have a facility for specifying floating-point numbers. The most common way is by a real declaration statement. High level programming languages must have a provision for handling floating-point arithmetic operations. The operations are generally built in the internal hardware. If no hardware is available, the compiler must be designed with a package of floating-point software subroutine. Although the hardware method is more expensive, it is much more efficient than the software method. Therefore, floating- point hardware is included in most computers and is omitted only in very small ones.

Basic Considerations:

There are two part of a floating-point number in a computer - a mantissa m and an exponent e. The two parts represent a number generated from multiplying m times a radix r raised to the value of e. Thus

$$m \times r^e$$

The mantissa may be a fraction or an integer. The position of the radix point and the value of the radix r are not included in the registers. For example, assume a fraction representation and a radix 10. The decimal number 537.25 is represented in a register with m = 53725 and e = 3 and is interpreted to represent the floating-point number

$.53725 \times 10^3$

A floating-point number is said to be normalized if the most significant digit of the mantissa in nonzero. So the mantissa contains the maximum possible number of significant digits. We cannot normalize a zero because it does not have a nonzero digit. It is represented in floating-point by all 0's in the mantissa and exponent.

Floating-point representation increases the range of numbers for a given register. Consider a computer with 48-bit words. Since one bit must be reserved for the sign, the range of fixed-point integer numbers will be $+ (24^7 - 1)$, which is approximately $+ 101^4$. The 48 bits can be used to represent a floating-point number with 36 bits for the mantissa and 12 bits for the exponent. Assuming fraction representation for the mantissa and taking the two sign bits into consideration, the range of numbers that can be represented is

$+ (1 - 2^{-35}) \times 2^{2047}$

This number is derived from a fraction that contains 35 1's, an exponent of 11 bits (excluding its sign), and because $2^{11}-1 = 2047$. The largest number that can be accommodated is approximately 10^{615}.

The mantissa that can accommodated is 35 bits (excluding the sign) and if considered as an integer it can store a number as large as $(2^{35} - 1)$. This is approximately equal to 10^{10}, which is equivalent to a decimal number of 10 digits.

Computers with shorter word lengths use two or more words to represent a floating-point number. An 8-bit microcomputer uses four words to represent one floating-point number. One word of 8 bits are reserved for the exponent and the 24 bits of the other three words are used in the mantissa.

Arithmetic operations with floating-point numbers are more complicated than with fixed-point numbers. Their execution also takes longer time and requires more complex hardware. Adding or subtracting two numbers requires first an alignment of the radix point since the exponent parts must be made equal before adding or subtracting the mantissas. We do this alignment by shifting one mantissa while its exponent is adjusted until it becomes equal to the other exponent.

Consider the sum of the following floating-point numbers: $.5372400 \times 10^2$

$+ .1580000 \times 10^{-1}$

Floating-point multiplication and division need not do an alignment of the mantissas. Multiplying the

two mantissas and adding the exponents can form the product. Dividing the mantissas and subtracting the exponents perform division.

The operations done with the mantissas are the same as in fixed-point numbers, so the two can share the same registers and circuits. The operations performed with the exponents are compared and incremented (for aligning the mantissas), added and subtracted (for multiplication) and division), and decremented (to normalize the result). We can represent the exponent in any one of the three representations - signed-magnitude, signed 2's complement or signed 1's complement.

Register Configuration(Registers for Floating Point arithmetic operations):

The register configuration for floating-point operations is shown in below figure. As a rule, the same registers and adder used for fixed-point arithmetic are used for processing the mantissas. The difference lies in the way the exponents are handled.

The register organization for floating-point operations is shown in below Fig. Three registers are there, BR, AC, and QR. Each register is subdivided into two parts. The mantissa part has the same uppercase letter symbols as in fixed-point representation. The exponent part may use corresponding lower-case letter symbol.

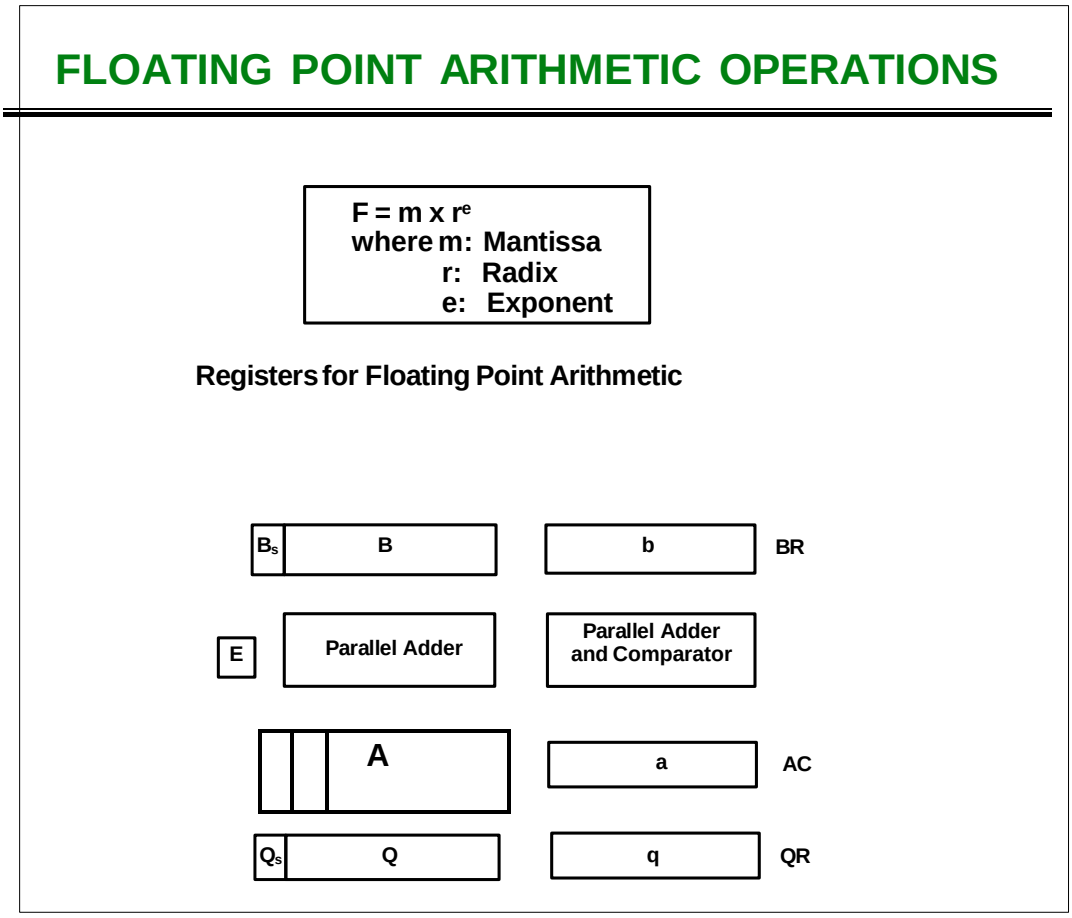


Figure : Registers for Floating Point arithmetic operations

Assuming that each floating-point number has a mantissa in signed-magnitude representation and a biased exponent. Thus the AC has a mantissa whose sign is in As, and a magnitude that is in A. The diagram shows the most significant bit of A, labeled by A1. The bit in his position must be a 1 to normalize the number. Note that the symbol AC represents the entire register, that is, the concatenation

of As, A and a.

In the similar way, register BR is subdivided into Bs, B, and b and QR into Qs, Q and q. A parallel-adder adds the two mantissas and loads the sum into A and the carry into E. A separate parallel adder can be used for the exponents. The exponents do not have a district sign bit because they are biased but are represented as a biased positive quantity. It is assumed that the floating- point number are so large that the chance of an exponent overflow is very remote and so the exponent overflow will be neglected. The exponents are also connected to a magnitude comparator that provides three binary outputs to indicate their relative magnitude.

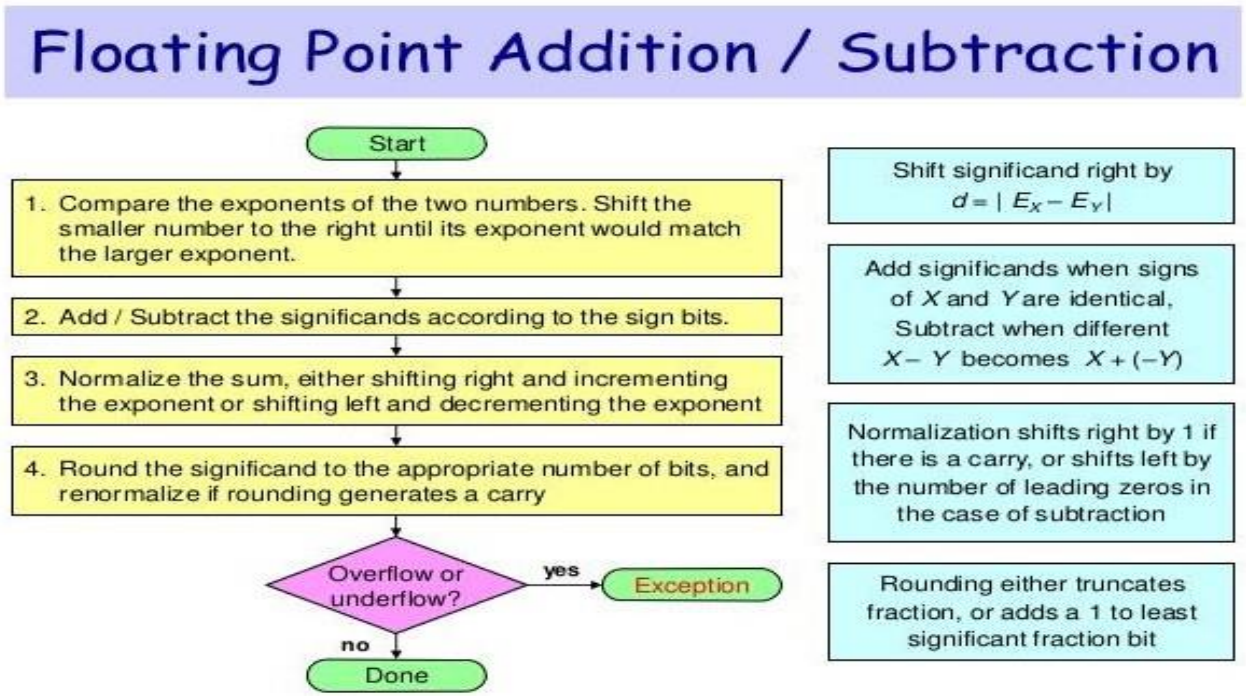
The number in the mantissa will be taken as a fraction, so they binary point is assumed to reside to the left of the magnitude part. Integer representation for floating point causes certain scaling problems during multiplication and division. To avoid these problems, we adopt a fraction representation.
The numbers in the registers should initially be normalized. After each arithmetic operation, the result will be normalized. Thus all floating-point operands are always normalized.

Addition and Subtraction of Floating Point Numbers:

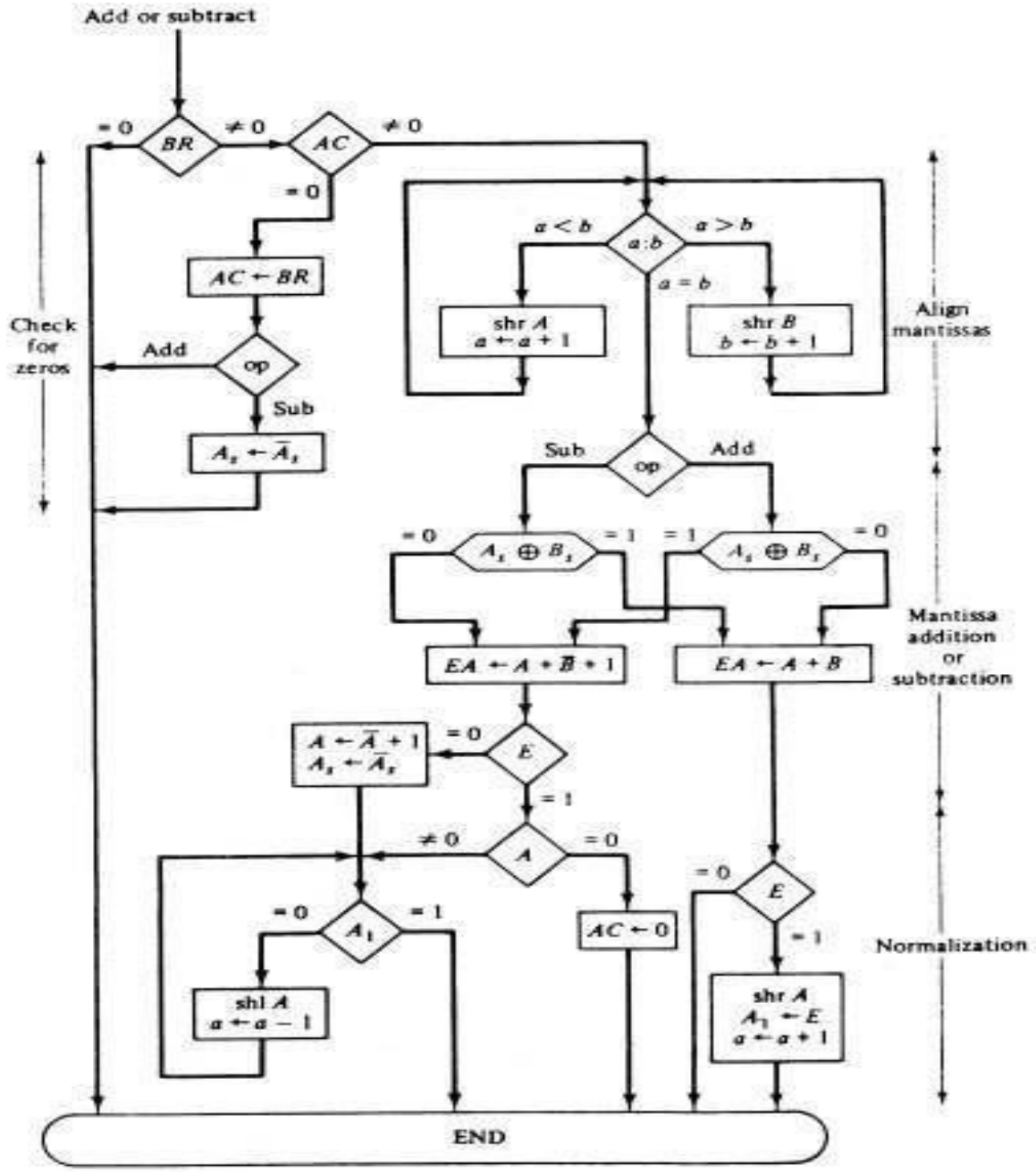
During addition or subtraction, the two floating-point operands are kept in AC and BR. The sum or difference is formed in the AC. The algorithm can be divided into four consecutive parts:

1. **Check for zeros.**
2. **Alignment of the mantissas.**
3. **Add or subtract the mantissas**
4. **Normalize the result**

A floating-point number cannot be normalized, if it is 0. If this number is used for computation, the result may also be zero. Instead of checking for zeros during the normalization process we check for zeros at the beginning and terminate the process if necessary. The alignment of the mantissas must be carried out prior to their operation. After the mantissas are added or subtracted, the result may be un-normalized. The normalization procedure ensures that the result is normalized before it is transferred to memory.
If the magnitudes were subtracted, there may be zero or may have an underflow in the result. If the mantissa is equal to zero the entire floating-point number in the AC is cleared to zero. Otherwise, the mantissa must have at least one bit that is equal to 1. The mantissa has an underflow if the most significant bit in position A1, is 0. In that case, the mantissa is shifted left and the exponent decremented. The bit in A1 is checked again and the process is repeated until A1 = 1. When A1 = 1, the mantissa is normalized and the operation is completed.



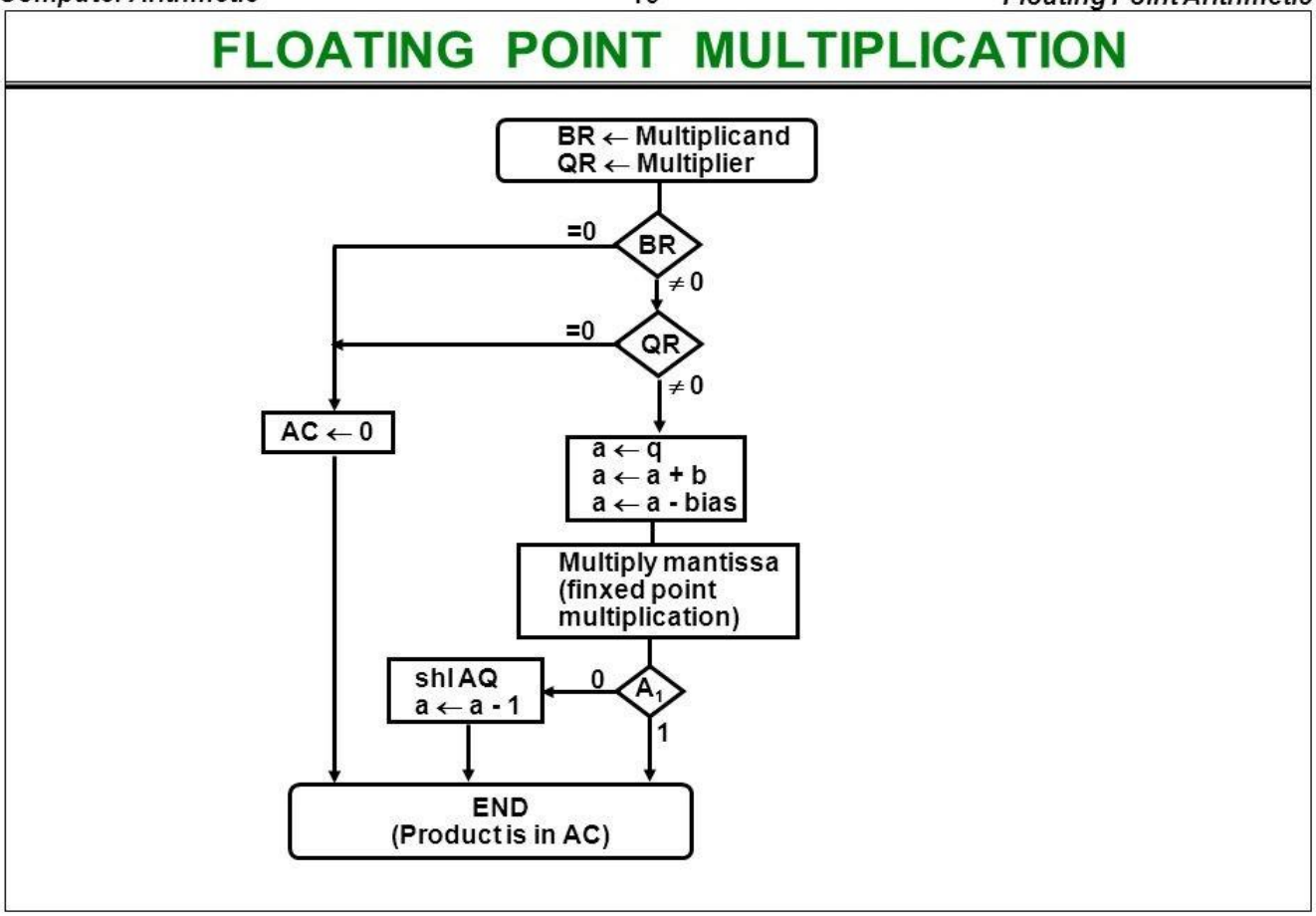
Algorithm for Floating Point Addition and Subtraction


Add or subtract
BR
AC
= 0
≠ 0
≠ 0
= 0
AC ← BR
Check for zeros
Add
op
Sub
A_s ← Ā_s
a < b
a : b
a > b
a = b
shr A
a ← a + 1
shr B
b ← b + 1
Align mantissas
Sub
op
Add
= 0
A_s ⊕ B_s
= 1
= 1
A_s ⊕ B_s
= 0
EA ← A + B̄ + 1
EA ← A + B
Mantissa addition or subtraction
A ← Ā + 1
A_s ← Ā_s
= 0
E
= 1
≠ 0
A
= 0
= 0
A_1
= 1
AC ← 0
= 0
E
= 1
shl A
a ← a − 1
shr A
A_1 ← E
a ← a + 1
Normalization
END

Floating Point Multiplication:

The algorithm can be divided into four consecutive parts:

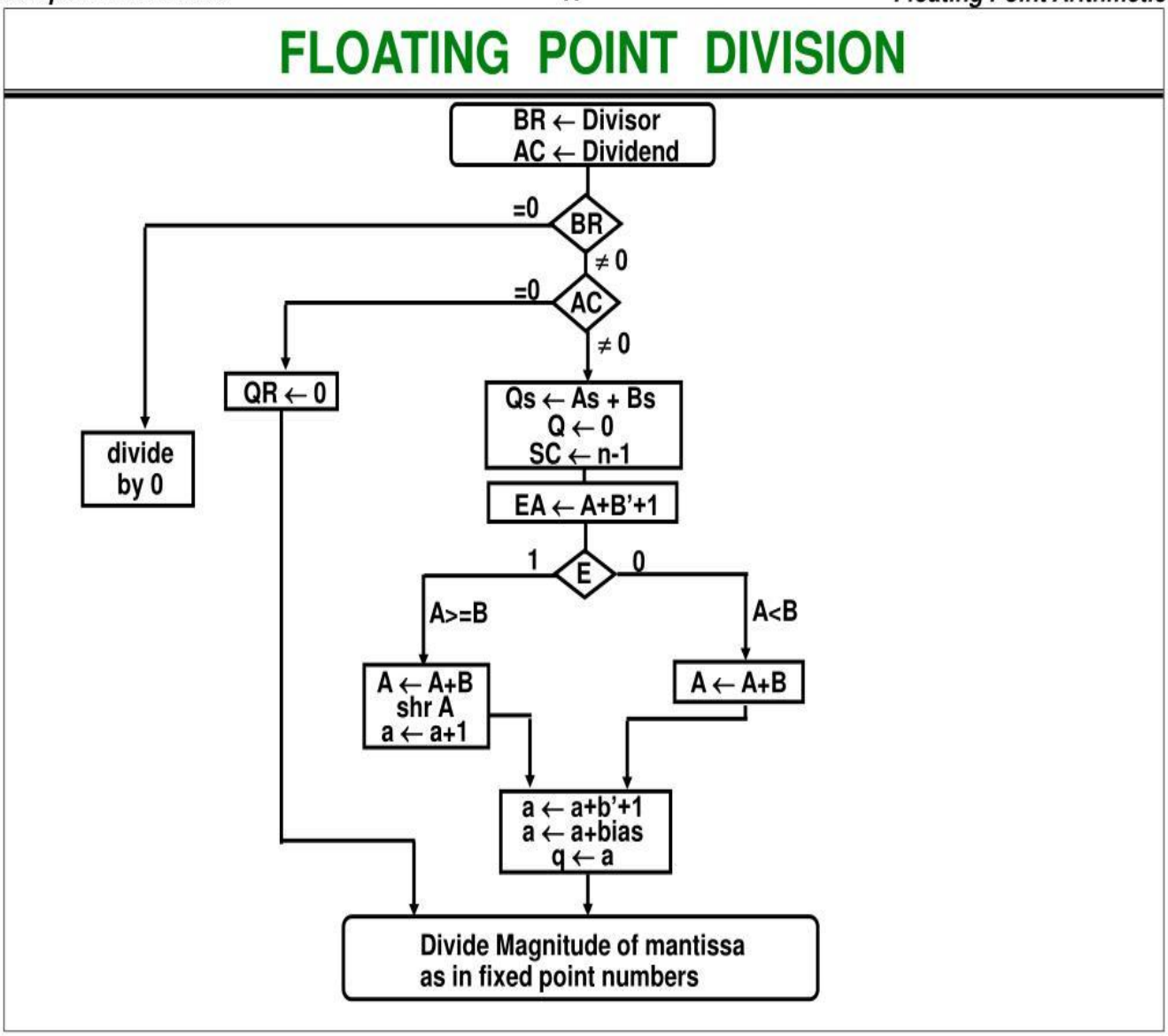
1. **Check for zeros.**
2. **Add exponents.**
3. **Multiply mantissas**
4. **Normalize the result**



Floating Point Division*:*

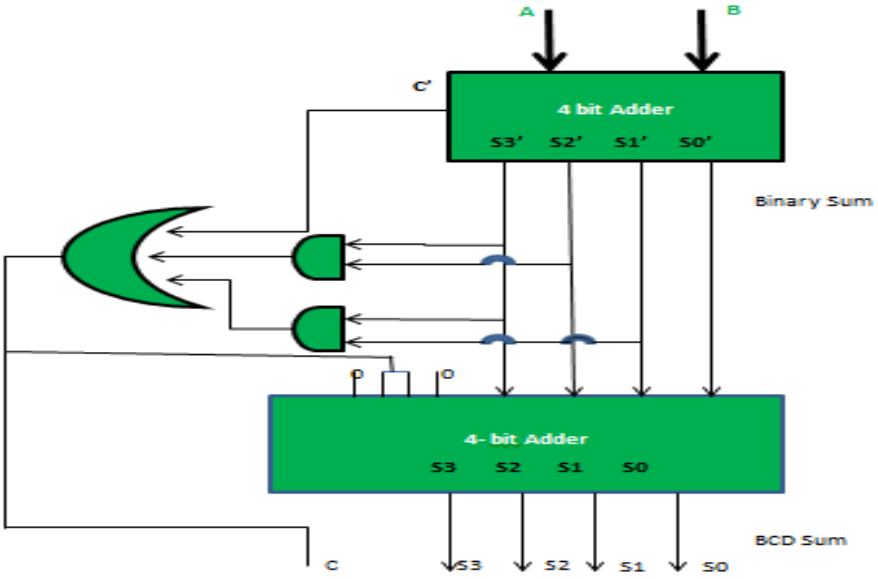
The algorithm can be divided into Five consecutive parts:

1. **Check for zeros.**
2. **Initialize the results and evaluate its sign**
3. **Align dividend or check overflow**
4. **Subtract the exponent**
5. **Divide the mantissas**



Decimal Arithmetic unit:

A decimal arithmetic unit for BCD addition and subtraction is built using a BCD adder and a method for subtraction, like the 9's or 10's complement. The core component is a 4-bit BCD adder, which adds two BCD digits. If the sum is greater than 9, a correction is made by adding 0110 (decimal 6) to the result to ensure it remains a valid BCD code. For subtraction, you can use a 9's complement or 10's complement of the subtrahend and then use the BCD adder to perform an addition.



BCD addition

1. **Use a 4-bit BCD adder**: This unit adds two 4-bit binary numbers, representing two BCD digits.
2. **Check the result**: After the addition, check if the sum is greater than 9 (binary 1001)
3. **Add the correction**: If the sum is greater than 9, it's an invalid BCD code. Add 0110 (decimal 6) to the sum to get the correct BCD output and generate a carry-out.
4. **Handle multiple digits**: For multi-digit numbers, connect multiple 4-bit BCD adders in series. The carry-out from one stage becomes the carry-in to the next.

BCD subtraction

1. **Use a mode bit**: A mode bit or control signal determines whether the unit performs addition or subtraction.
2. **Use 9's complement**: For subtraction, find the 9's complement of the subtrahend and add it to the minuend using the BCD adder.
   9's complement: Invert each bit of the 4-bit BCD number
   (e.g., 0101 becomes 1010)
3. **Use 10's complement**: Alternatively, find the 10's complement of the subtrahend and add it to the minuend.
   - **10's complement**: Find the 9's complement, then add 1.
4. **Handle the final carry**: In the 10's complement method, if a final carry is generated, discard it. If no carry is generated, the result is negative, and its 9's complement represents the magnitude of the result, with the sign indicated separately.

Decimal Arithmetic operations:

Decimal arithmetic operations are performed on numbers using a decimal arithmetic unit, which handles the conversion of decimal numbers to their binary-coded decimal (BCD) representation, performs operations like addition, subtraction, multiplication, and division, and then converts the binary result back into decimal. Key operations include using a BCD adder for addition and a BCD subtractor for subtraction, with the latter often involving complements like the 10's complement.

Key components and operations

1. **Decimal Arithmetic Unit:**
   A digital function that processes decimal numbers. Because computers work with binary, this unit handles the conversion to and from BCD, as well as the arithmetic operations.
2. **Binary-Coded Decimal (BCD)**:
   The format used to represent decimal digits. Each decimal digit $(0-9)$ is stored as a 4-bit binary code.
3. **Addition:**
   Performed using a BCD adder. A key aspect is adjusting the result if the sum of two 4-bit groups exceeds 9(decimal), which is done by adding 6(binary 0110) to the sum and generating a carry.
4. **Subtraction:**
   Typically done using the 10's complement method. This involves finding the 9's complement of the subtrahend (inverting the bits) and then adding 1.
5. **Multiplication and Division**:
   These operations are also performed by the decimal arithmetic unit. The algorithms are similar to binary counterparts but require specific adjustments to handle BCD and the 4-bit groups representing each decimal digit.
6. **Shift operations**:
   These move the digits within registers.

- **Decimal Shift Right (DSHR)**: Shifts all digits one position to the right, discarding the rightmost digit and filling the leftmost position with a 0
- **Decimal Shift Left (DSHL)**: Shifts all digits one position to the left, discarding the leftmost digit and filling the rightmost position with a 0

The table shows symbols for decimal arithmetic micro-operations.

Symbols for Decimal Arithmetic Micro-Operations

Symbolic Representation	Meaning
$X \leftarrow X + Y$	It can add decimal numbers and transfers the output to X.
Y'	9's complement of Y.
$X \leftarrow X + Y' + 1$	It can add the content of X and 10's complement of Y and transfers the output to X.
dshr X	It can shifts the decimal number one digit towards the right in register X.
dshl X	It can shifts the decimal number one digit towards left in register X

In this table, we can see a bar over the symbol for the register letter. This refers to the 9's complement of decimal number that is stored in the register. When 1 is added to the 9's complement the 10's complement is produced.

Therefore, the symbol $X \leftarrow X+ Y + 1$ for decimal digits denotes, transfer of decimal sum that was formed by adding the original content X to the 10's complement of Y.

It may be confusing to use similar symbols for 9's complement and 1's complement in case both types of data are used in the same system.

Therefore, it would be better to implement a different symbol for the 9's complement. In case only one type of data is taken into consideration, the symbol would apply to the type of data used.