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:

Binary information in digital computers is stored in memory or processor registers.

• 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.

• A number system of base or radix r is a system of that uses distinct symbols for r digits

• The decimal number system in everyday use employs radix 10 system. The 10 symbols are 0, 1, 2, 3, 4, 5, 6,7, 8, and 9; highest number being r-1

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 1 x 25 + 0 x 24 + 1 x 23 + 1 x 22 + 0 x 21 + 1 x 20 = 45

• Besides the decimal and binary number systems,

• Octal (radix 8)- 0,1,2,3,4,5,6,7 and

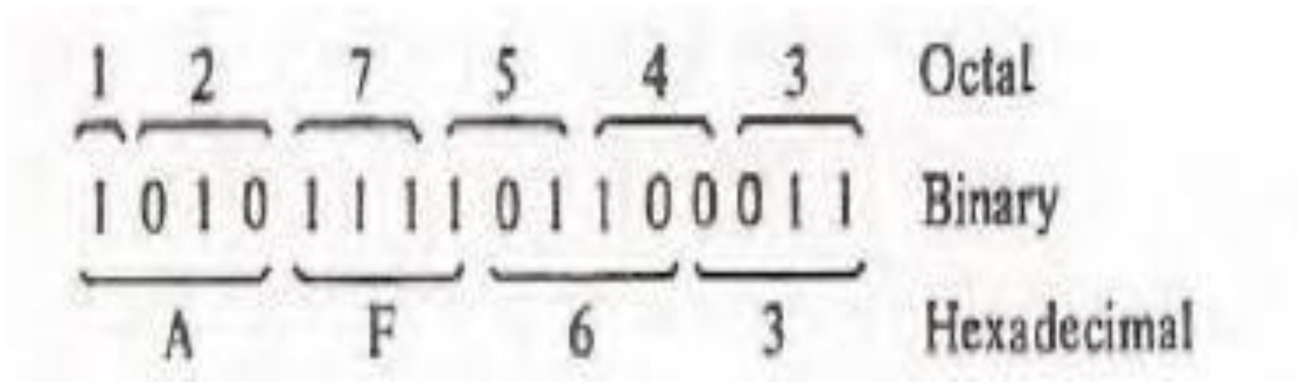
• Hexadecimal (radix 16) – 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

• Octal can be converted to decimal as follows

(736.4)8 = 7 X 82 + 3 X 81 + 6 X 80 + 4X8-1 = 7X64 + 3X8 + 6X1 + 4/8 = (478.5) 10

Hexa deciamal can be converted to decimal (F3)16 = F x 16 + 3 = 15 x 16 + 3 = (243)10

Octal and Hex can be obtained from Binary as shown below:



Complements

A binary code is a group of n bits that assume upto 2^n distinct combinations of 0s and 1s.

A BCD code is a binary coded decimal i.e. binary coding decimal numbers.

ASCII (American Standard Code for Information Interchange), which uses seven bits to code 128 characters is standard alphanumeric character code.

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 - 1)'scomplement

• For binary base 2 system: the 2's complement and the 1's complement
• For decimal base 10 system: the 10's complement and the 9's complement
• The 9's complement of 546700 is 999999 - 546700 = 453299
• The 9's complement of 12389 is 99999 - 12389 = 87610
• The 1's complement of a binary number is formed by Changing 1's into 0's and 0's into 1's.
• For example, the 1's complement of 1011001 is 0100110 and the 1's complement of 0001111 is 1110000.

The 10's complement of the decimal 2389 is 7610 + 1 = 7611 and is obtained by adding 1 to the 9's complement value.

• The 2's complement of binary 101100 is 010011 + 1 = 010100 and is obtained by adding 1 to the 1's complement value.

• For example, the 1's complement of 1011001 is 0100110 and the 1's complement of 0001111 is 1110000.

• Solve: Find the 9’s and 10’s complement of 246700. Find the 1’s and 2’s complement of 1101100.

Fixed-Point Representation

- In computer binary systems it is customary to represent the sign with a bit placed in the leftmost position of the number.
- sign bit is equal to 0 for positive and to 1 for negative.
- a number may have a binary (or decimal) point.
- There are two ways of specifying the position of the binary point: by giving it a fixed position or by employing a floating- point representation.
- The fixed-point method assumes that the binary point is always. fixed in one position.
- The two positions most widely used are (1) a binary point in the extreme left of the register to make the stored number a fraction, and (2) a binary point in the extreme right of the register to make the stored number an integer. In either case, the binary point is not actually present.

Fixed-Point Representation

Integer Representation for signed numbers

- signed-magnitude representation 1 0001110
- signed-1's complement representation 1 1110001
- signed-2's complement representation 1 1110010

Floating Point Representation

- The floating-point representation uses a second register to store a number that designates the position of the decimal point in the first register.
- The floating-point representation of a number has two parts. The first part represents a signed, fixed-point number called the mantissa. The second part designates the position of the decimal (or binary) point and is called the exponent The fixed- point mantissa may be a fraction or an integer.

For example,

- the decimal number +6132.789 is represented in floating-point with a fraction and an exponent as follows:

Fraction	*Exponent*
+0.6132789	*+04*

Floating Point Representation

A floating-point binary number is represented in a similar manner except that it uses base 2 for the exponent.
For example, the binary number +1001.11 is represented with an 8-bit fraction and 6-bit exponent as follows:

Fraction	Exponent
01001110	000100

COMPUTER ARITHMETIC

Introduction:

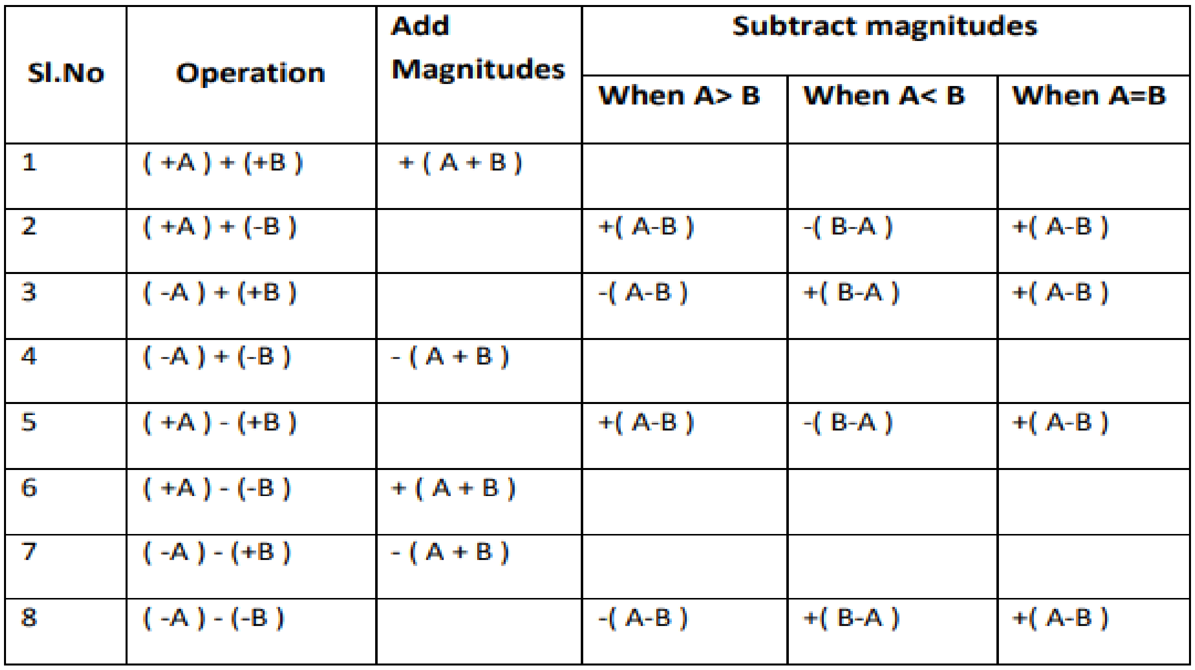
Data is manipulated by using the arithmetic instructions in digital computers. Data is manipulated to produce results necessary to give solution for the computation problems.
The Addition, subtraction, multiplication and division are the four basic arithmetic operations.
Using these operations other arithmetic functions can be formulated and scientific problems can be solved by numerical analysis methods.

Addition and subtraction algorithm for signed-magnitude data:

The representation of numbers in signed-magnitude is familiar because it is used in arithmetic calculations.

- Let the magnitude of two numbers be A & B.
- When signed numbers are added or subtracted, there are 4 different conditions to be considered for each addition and subtraction depending on the sign of the numbers.

- The conditions are listed in the table below. The table shows the operation to be performed with magnitude(addition or subtraction) are indicated for different conditions



Sl.No	Operation	Add Magnitudes	Subtract magnitudes		
			When A> B	When A< B	When A=B
1	(+A) + (+B)	+ (A + B)			
2	(+A) + (-B)		+(A-B)	-(B-A)	+(A-B)
3	(-A) + (+B)		-(A-B)	+(B-A)	+(A-B)
4	(-A) + (-B)	- (A + B)			
5	(+A) - (+B)		+(A-B)	-(B-A)	+(A-B)
6	(+A) - (-B)	+ (A + B)			
7	(-A) - (+B)	- (A + B)			
8	(-A) - (-B)		-(A-B)	+(B-A)	+(A-B)

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.

- The algorithm for addition and subtraction (from the table above):

Addition Algorithm:

- When the signs of A and B are identical, add two magnitudes and attach the sign of A to the result.
- When the sign 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 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 two magnitudes and attach the sign of A to the result.

When the sign 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 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 Implementation:

- Let A and B are two registers that hold the numbers. AS and BS are 2, flip- flops that hold sign of corresponding numbers.
- The result is stored in A and AS and thus they form Accumulator register.
- We need to perform micro operation, A+ B and hence a parallel adder is required.
- A comparator is needed to establish if A> B, A=B, or A=B, or A<B
- We need to perform micro operations A-B and B-A and hence two parallel subtractor are required.
- An exclusive OR gate can be used to determine the sign relationship, that is, equal or not.
- Thus the hardware components required are a magnitude comparator, an adder, and two subtractors

Reduction of hardware by using different procedure:

- We know subtraction can be done by complement and add.
- The result of comparison can be determined from the end carry after the subtraction.
- We find an adder and a complementer can do subtraction and comparison if 2’s complement is used for subtraction.

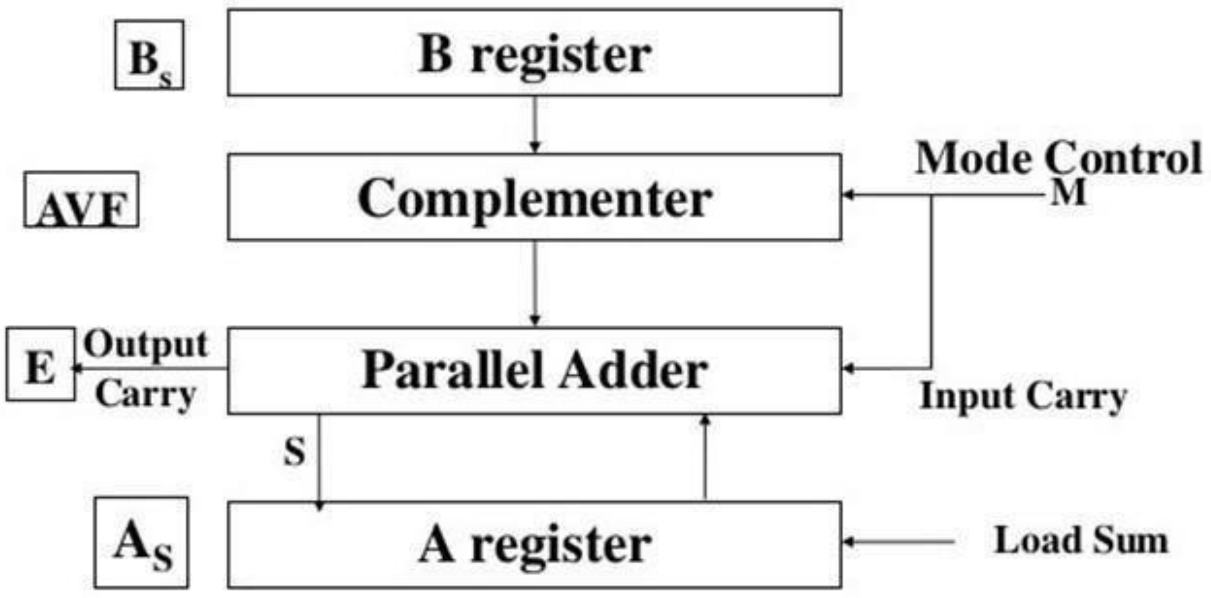


Figure 1: Hardware for signed-magnitude addition and subtraction

AVF Add overflow flip flop. It hold the overflow bit when A & B are added.

- **Flip flop E** —Output carry is transferred to E. It can be checked to see the relative magnitudes of the two numbers.
- A-B = A +(-B)= Adding A and 2's complement of B.
- The A register provides other micro operations that may be needed when the sequence of steps in the algorithm is specified.
- The complementer passes the contents of B or the complement of B to the Parallel Adder depending on the state of the mode control M.
- It consists of EX-OR gates and the parallel adder consists of full adder circuits.
- The M(Mode Control) signal is also applied to the input carry of the adder. When input carry **M=0, the sum of full adder is A +B.**
- When M=1, **S = A + B' +1= A – B**

Hardware algorithm: Flow Chart for Add and Subtract operations:

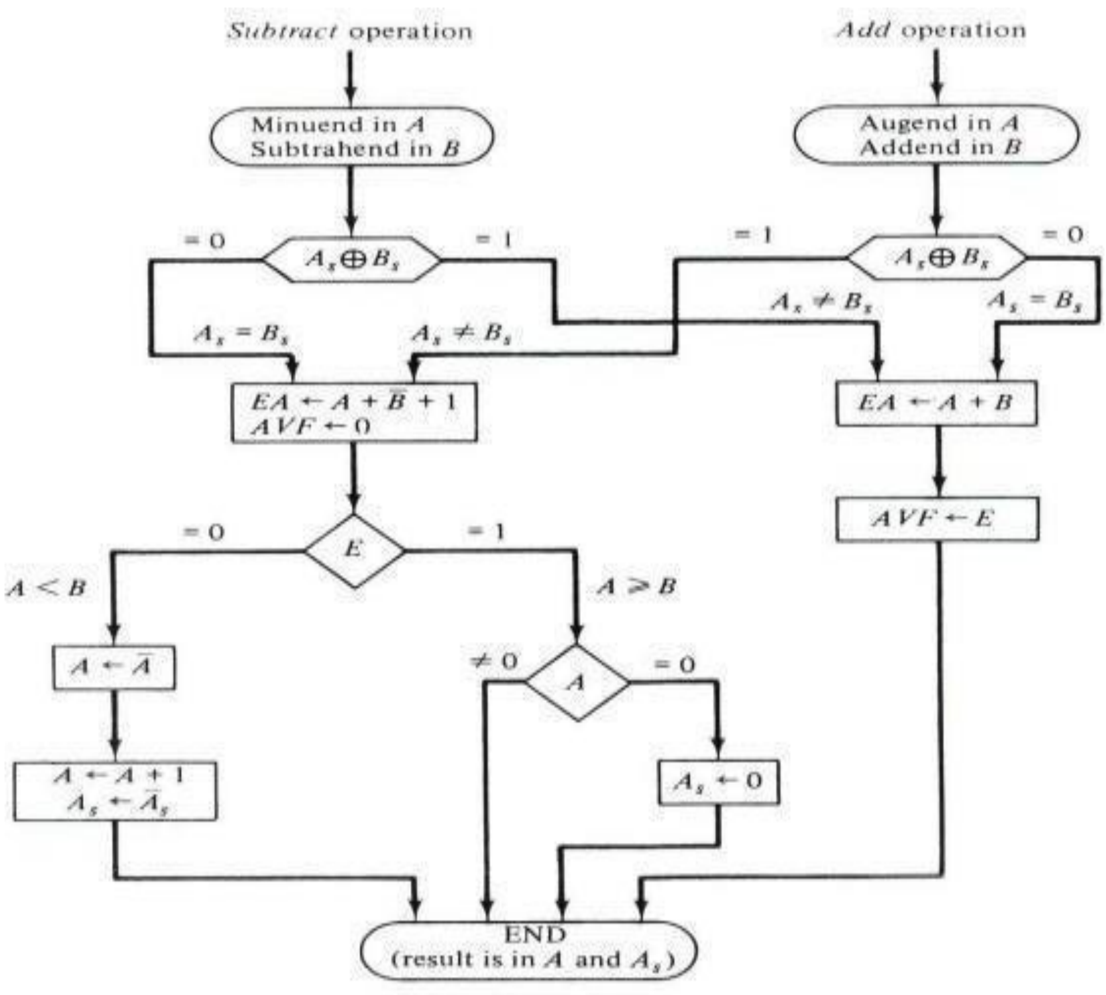


Figure 10-2 Flowchart for add and subtract operations.

- The EX-OR gate provides 0 as output when the signs are identical. It is 1 when the signs are different.
- **A + B is computed for the following and the sum is stored in EA:**
  1. When the signs are same and addition operation is required.
  2. When the signs are different and subtract operation is required.
- The carry in E after addition indicates an overflow if it is 1 and it is transferred to AVF, the add overflow flag
- **A-B = A+ B'+1 computed for the following:**
- 1. When the signs are different and addition operation is required.

- 2. When the signs are same and subtract operation is required. No overflow can occur if the numbers are subtracted and hence AVF is cleared to Zero.
- **A 1 in E indicates that A ≥ B** and the number in A is the correct result. If this number in A is zero, the sign AS must be made positive to avoid a negative zero.
- **A 0 in E indicates that A< B**. For this case it is necessary to take the 2’s complement of the value in A.
- In the algorithm shown in flow chart, it is assumed that A register has circuits for micro operations complement and increment.
- Hence two complement of value in A is obtained in 2, micro operations..
- In other paths of the flow chart, the sign of the result is the same as the sign of A, so no change in AS is required.
- **However, when A < B, the sign of the result is the complement of original sign of A.**
- Hence the **complement of AS stored in AS.**
- Final Result: As and A

Addition and Subtraction with signed-2’s complement Data

- The addition of two numbers in signed-2's complement form consists of adding the numbers with the sign bits treated the same as the other bits of the number.
- A carry-out of the sign-bit position is discarded.
- The subtraction consists of first taking the 2's complement of the subtrahend and then adding it to the minuend.
- The register configuration for the hardware implementation is shown in Figure below.

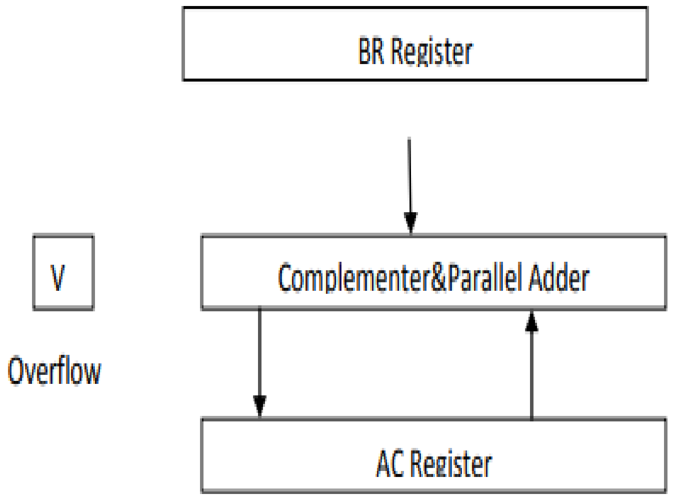
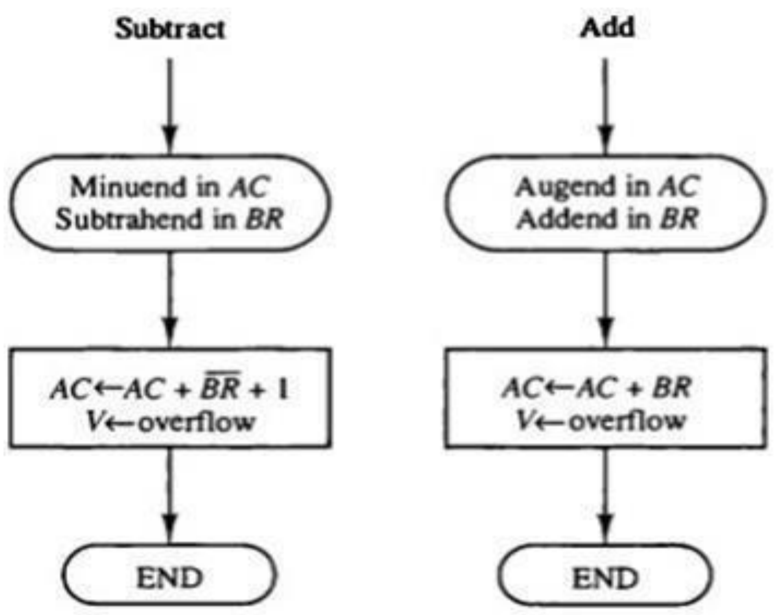


Fig: Hardware for Signed 2/s complement for addition/ subtractioin.

Hardware implementation of signed 2’s complement for addition/subtraction

- Here the sign bits are not separated from the registers and named it as AC(Accumulator) and the B register(BR)
- The leftmost bit in AC and BR represents the sign bits of the numbers.
- The two sign bits are added or subtracted together with the other bits in the complementer and parallel adder.
- The overflow flip-flop V is set to1 if there is an overflow.
- The output of the carry in this case is discarded.
- The algorithm for adding and subtracting two binary numbers in signed2's complement representation is shown in the flow chart below

Algorithm for adding and subtracting numbers in 2’s complement form:



Algorithm for adding and subtracting numbers in 2's complement form

- The sum is obtained by adding the contents of AC and BR (including their sign bits).
- 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.
- 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.
- 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.

Multiplication Algorithms:

- Multiplication of two fixed-point binary numbers in signed-magnitude representation is done with process of successive shift and adds operations.

- This process is best illustrated with a numerical example as follows:

```
 23        10111    Multiplicand
 19      × 10011    Multiplier
         -------
           10111
          10111
         00000     +
        00000
       10111
       ---------
437    110110101    Product
```

Numerical example of Multiplication

- If the multiplier bit is equal to 1, the multiplicand is copied down; otherwise zeros are copied down.
- The numbers copied down are shifted one position to the left from the previous number.
- Finally, the numbers are added and their sum forms the product.

Hardware Implementation for Signed-Magnitude Data Multiplication:

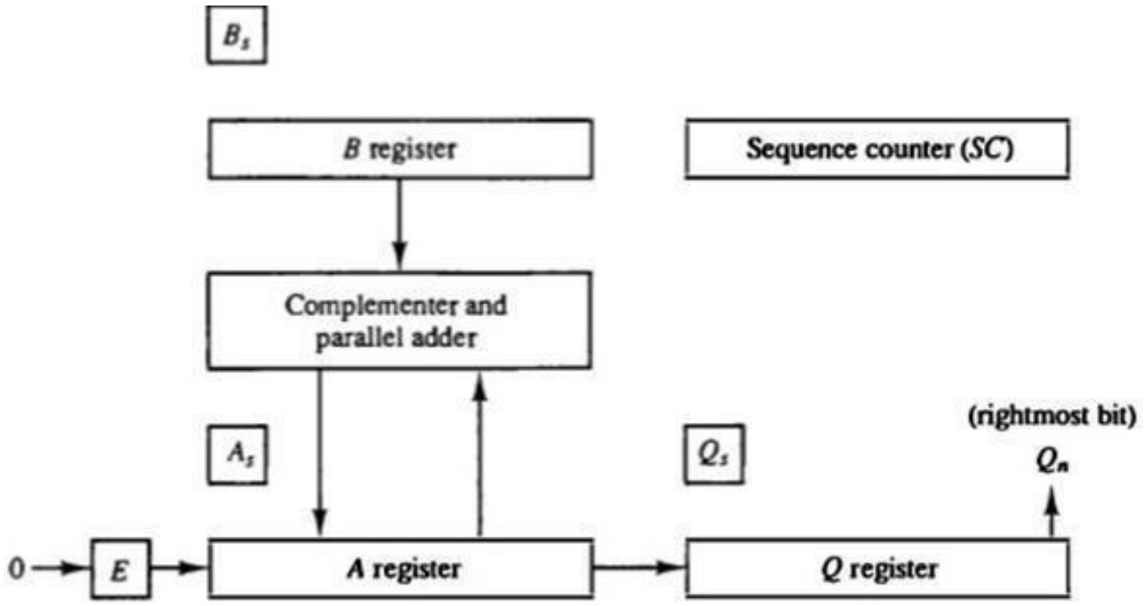


Figure 5: Hardware for multiply operation

- The hardware for multiplication consists of the equipment shown in Figure above.
- Initially, the multiplicand is in register B and the multiplier in Q.
- Their corresponding signs are stored in the flip-flops Qs and Bs
- Initially A is set to 0 as number of bits in the multiplicand.
- The **sequence counter SC** is initially set to a number equal to the number of bits in the multiplier.
- The sum of A and B forms a partial product which is transferred to the EA register.
- Both partial product and multiplier are **shifted to the right.**
- This shift will be denoted by the statement **shr EAQ** to designate the right shift depicted in Figure above.
- The least significant bit of A is shifted into the most significant position of Q, the bit from E is shifted into the most significant position of A, and 0 is shifted into E.
- After the shift, one bit of the partial product is shifted into Q, pushing the multiplier bits one position to the right.
- In this manner, the rightmost flip-flop in register Q, designated by Q_n, will hold the bit of the multiplier, which must be inspected next.
- The **counter is decremented by 1 after forming each partial product**. When the content of the counter reaches zero, the product is formed and the process stops.

Hardware Algorithm(Flow chart) Signed-Magnitude Data Multiplication:

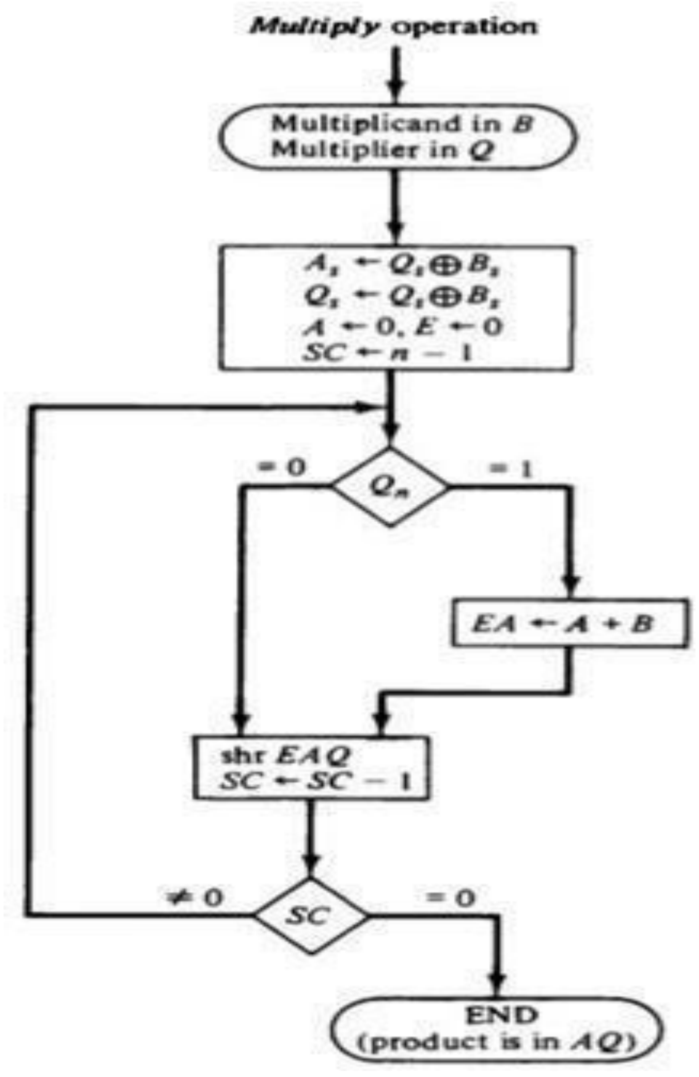


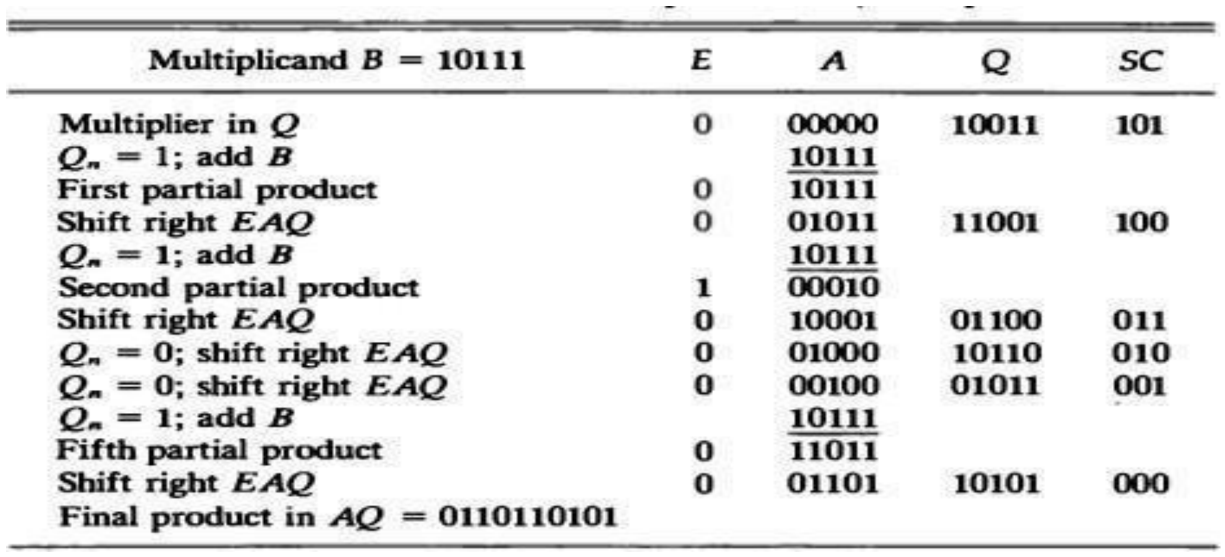
Fig : Flowchart multiply operation on sign magnitude representation numbers

- Initially, the multiplicand is in B and the multiplier in Q. Theircorresponding signs are in B_s and Q_s, respectively.
- **The signs are compared**, and both signs of 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.
- 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.

- After the initialization, the low-order bit of the multiplier in Qn, is tested.
- If **Qn is a 1**, the multiplicand in B is added to the present partial product in A. If Qn is a 0, nothing is done.
- 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.
- Note that 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.
- The following table describes multiplication of binary numbers 10111(+23) and 10011(+19) which are represented using Sign Magnitude Representation.

Table : Numerical Example for Binary Multiplier

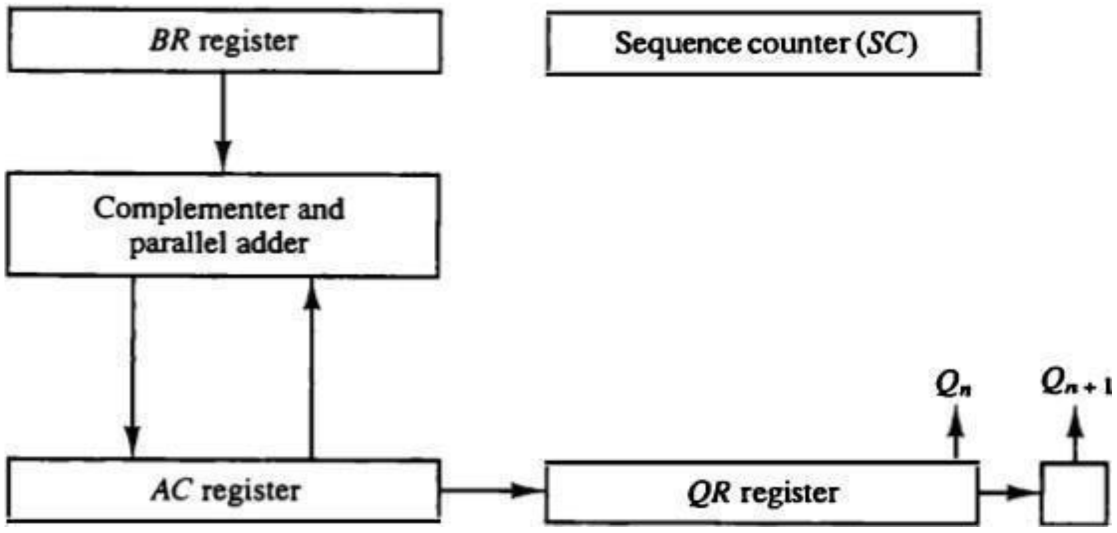


Multiplicand B = 10111	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
$Q_n = 1$; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
$Q_n = 1$; add B		10111		
Second partial product	1	00010		
Shift right EAQ	0	10001	01100	011
$Q_n = 0$; shift right EAQ	0	01000	10110	010
$Q_n = 0$; shift right EAQ	0	00100	01011	001
$Q_n = 1$; add B		10111		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000
Final product in AQ = 0110110101				

- Now Result is available in Registers A and Q. i.e. 0110110101 => 437 and sign bit of A is 0. So result is +437.
- The following table 3 describes multiplication of binary numbers **10011(+19)** and **00110(+6)**which are represented using Sign Magnitude Representation.
- Here Multiplicand is positive value, so $B_s = 0$. Here Multiplier is positive value, so $Q_s = 0$.
- Now $A_s = B_s + Q_s$, i.e A_s is positive; when both Bs and Qs are equal

Booth Multiplication Algorithm (for signed-2's complement numbers)

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



- 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.

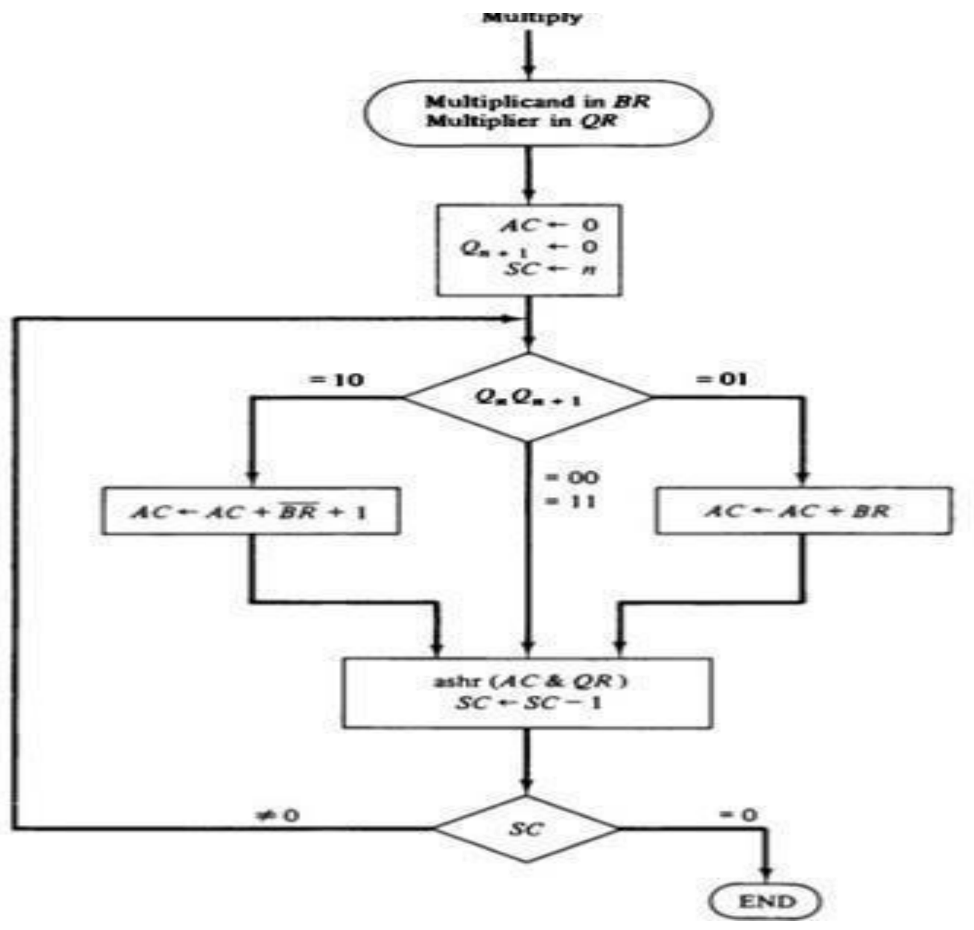
2. The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous 1) in a string of 0's in the multiplier.

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

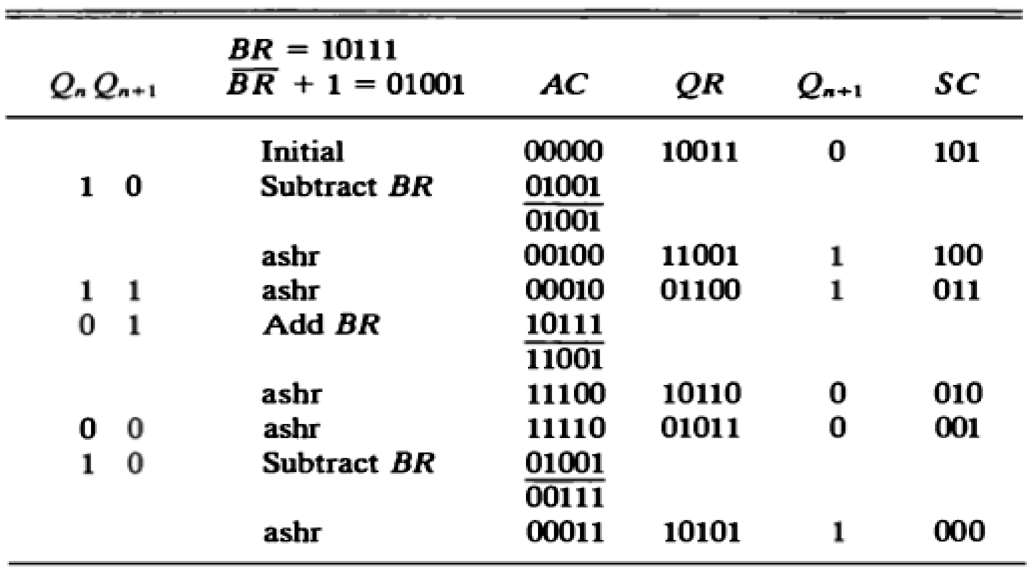
The hardware implementation of Booth algorithm requires the register configuration shown in Figure.

Q_n designates the least significant bit of the multiplier in register QR. An extra flip-flop $Q_{n+1}$is appended to QR to facilitate a double bit inspection of the multiplier.

The flowchart for Booth algorithm is shown in Figure .



- 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.
- **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 thepartial product in AC.
- When the two bits are equal, the partial product does not change. An *overflow cannot* occur because the addition and subtraction of the multiplicand follow each other.
- 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.
- A numerical example of Booth algorithm is shown in Table 5. It shows the step-by-step multiplication of (-9) x (-13) = + 117.
- Here the multiplier in QR is negative and that the multiplicand in BR is also negative. The 10-bit product appears in AC and QR and is positive.



$Q_n Q_{n+1}$	$BR = 10111$ $\overline{BR} + 1 = 01001$	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	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

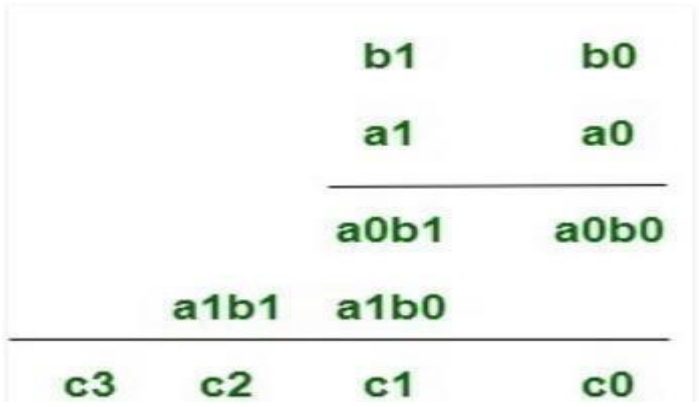
Table : Example of Multiplication with Booth Algorithm

- Now Result is available in Registers AR and QR. i.e. 0001110101 =>+117.

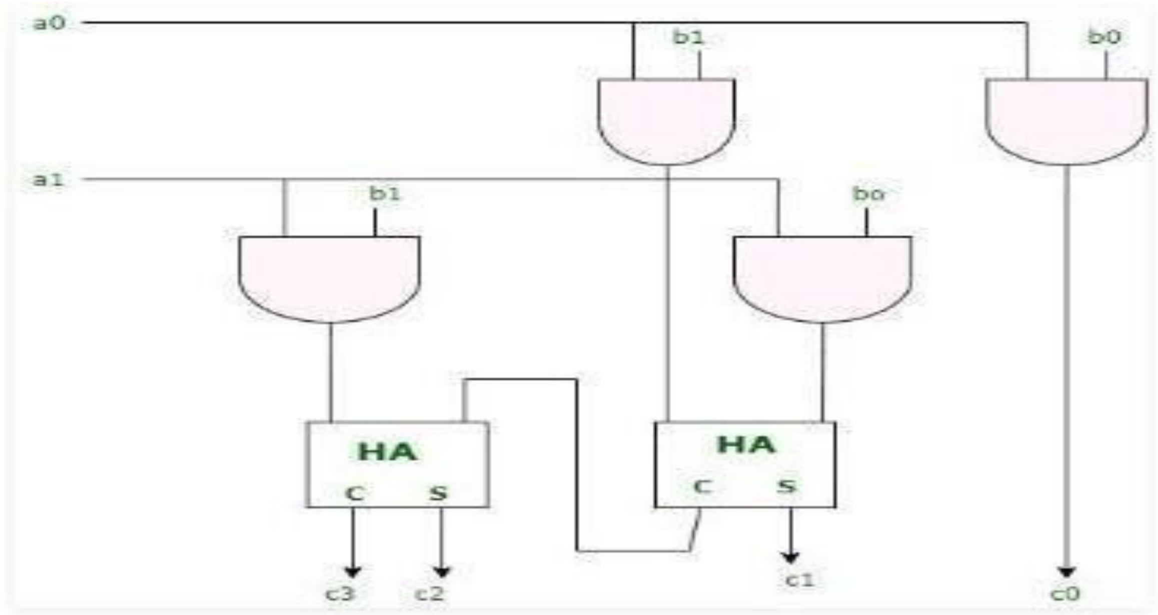
ARRAY MULTIPLIER::

- An Array multiplier is implemented with **combinational circuit.**
- Consider the multiplication of two 2-bit numbers as shown in figure.
- The multiplicand bits are **b1 and bo**; the multiplier bits are **a1 and a0** and the product is c3c2c1c0.
- The partial product is formed by multiplying a0 by b1b0.
- The multiplication of two bits such as ao and b0 produces a result 1 if both bits are 1; otherwise , it produces a 0.
- This is identical to an AND operation and can be implemented with an AND gate.
- As shown in the figure, the first partial product is formed by means of two AND gates.
- The second partial product is formed by multiplying a1 by b1b0 and is shifted to one position to the left.
- The two partial products are added with **two half adders circuits.**

2 bit by 2 bit Array multiplier



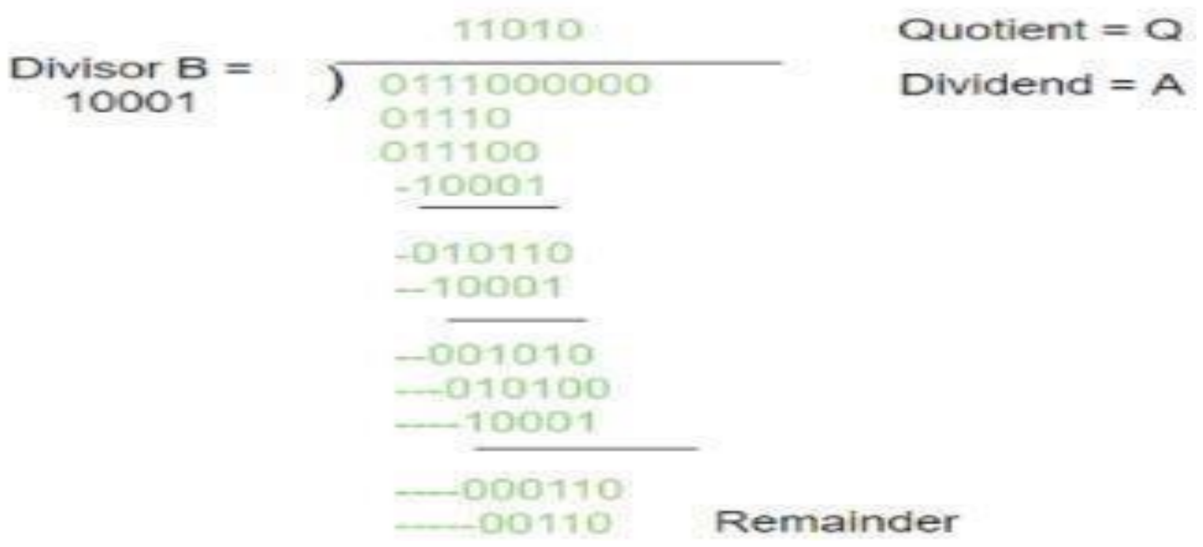
		b1	b0
		a1	a0
		a0b1	a0b0
	a1b1	a1b0	
c3	c2	c1	c0



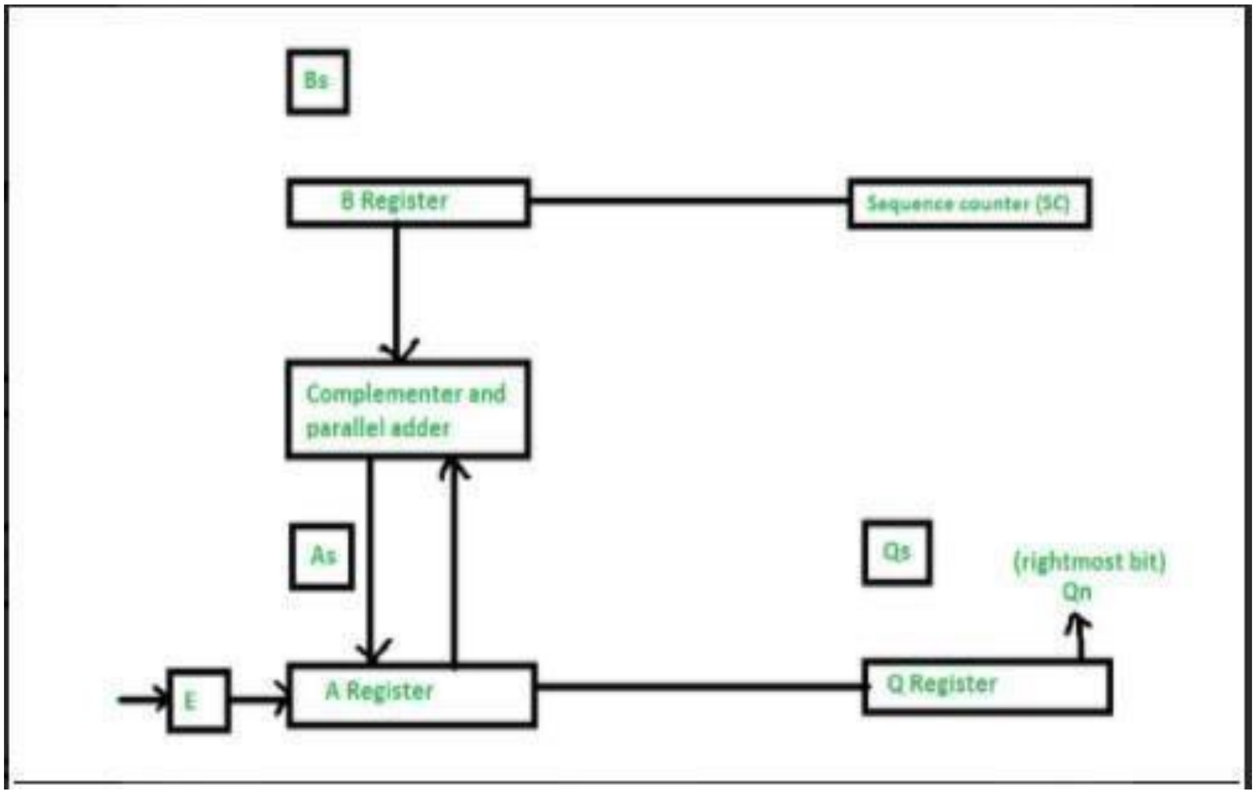
Division Algorithm:

- 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.
- The division process is described in Figure

Example of Division Operation:



Hardware Implementation

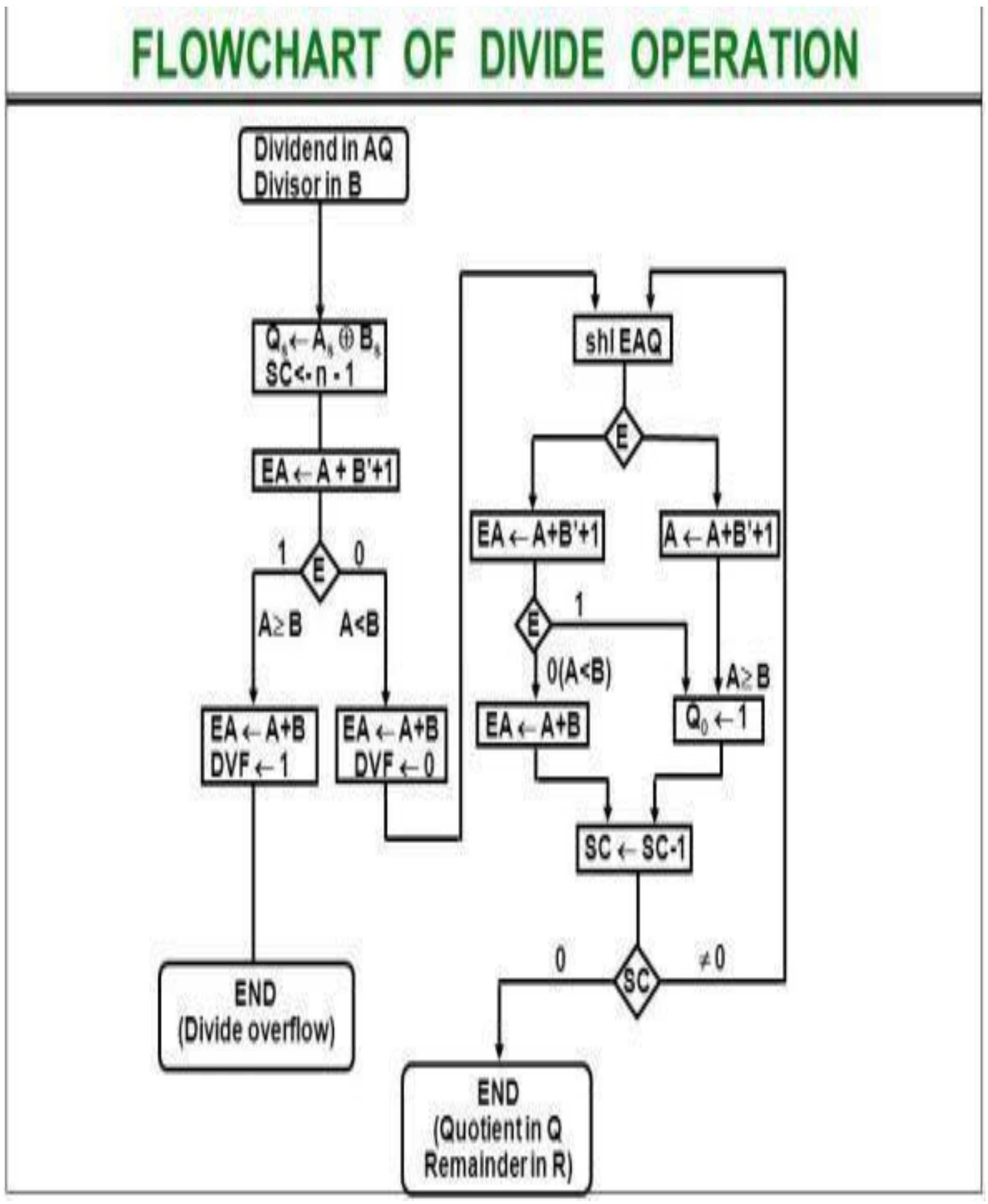


Division Operation using Pen and Paper:

- The divisor 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.

Hardware Implementation for Signed-Magnitude Data

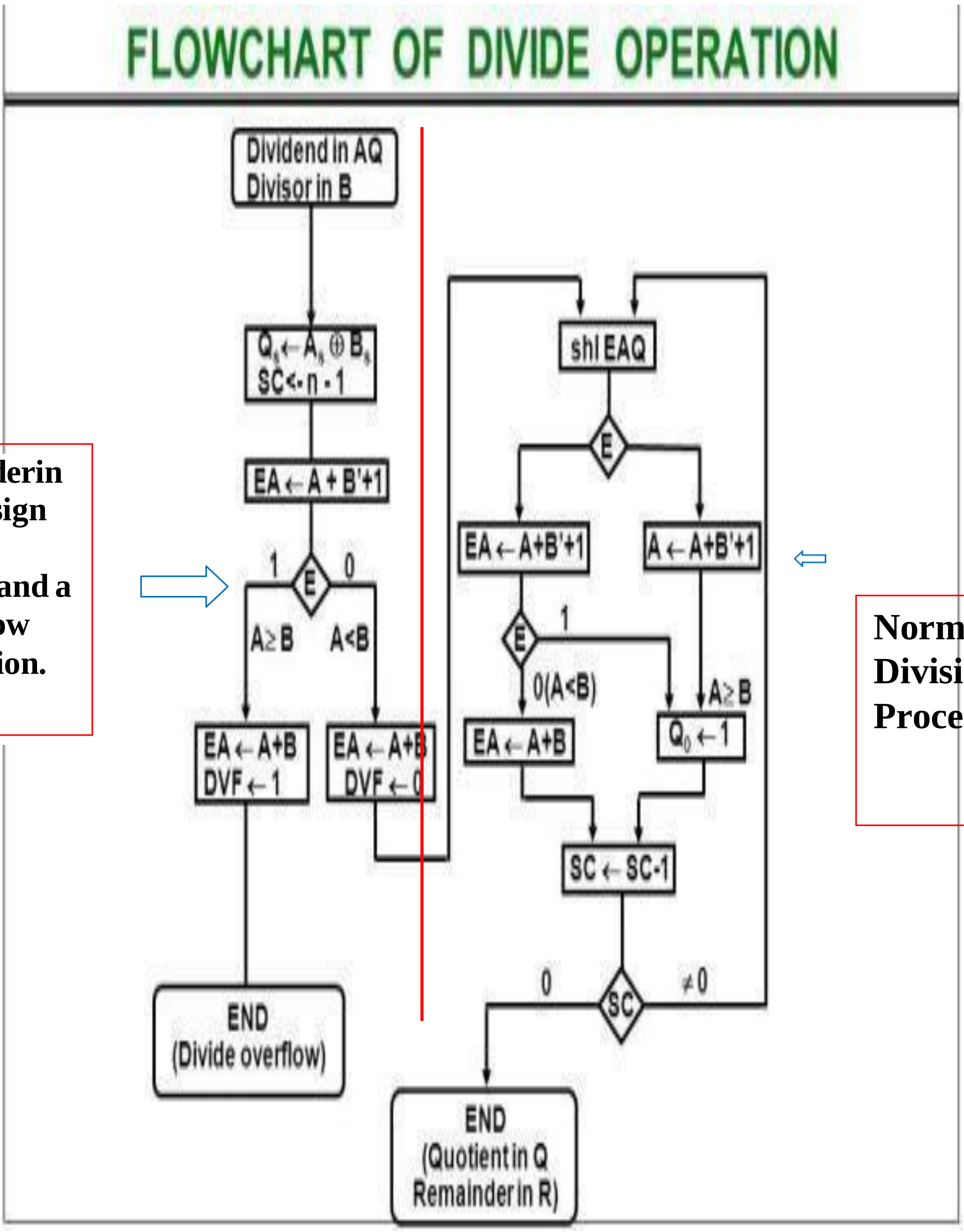
- In hardware implementation for signed-magnitude data in a digital computer, it is convenient to change the process slightly.
- Instead of shifting the divisor to the right, two dividends, or **partial remainders, are shifted to the left,** thus leaving the two numbers in the required relative position.
- 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.



- 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.
- **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 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.
- End carry(E) gives the information about the relative magnitudes.
- If E = 1, it signifies that A ≥ B. The quotient bit 1 is inserted into Qn and the partial remainder is shifted to left to the process.
- If E = 0, it signifies that A < B. So the quotient in Qn remains a 0.
- The value of B is added to restore the partial remainder in A to restore to its previous value.
- The partial remainder is shifted to the left and the process is repeated again until all quotient bits are formed.
- The remainder is then found in register A and the quotient is in register Q.
- Before showing the algorithm in flowchart form, we have to **consider the sign of the result and a overflow condition.**

Considering the sign of the result and a Overflow condition.



Considering the sign of the result and a overflow condition.

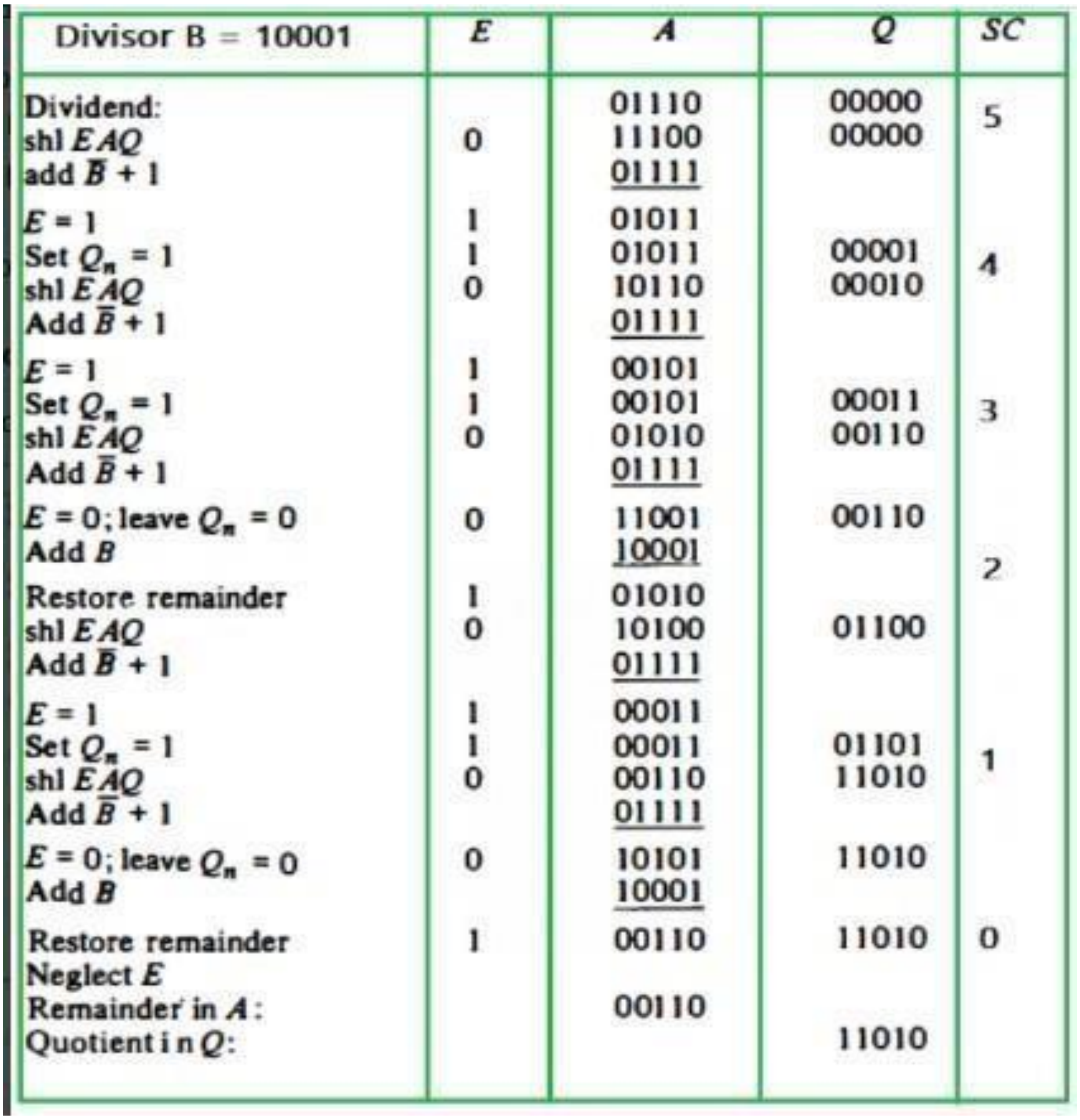
Normal Division Process

- Initially, the dividend is in A & Q and the divisor is in B.
- Sign of result is transferred into Q, to be the part of quotient. Then a constant is set into the SC to specify the number of bits in the quotient.
- Since an operand must be saved with its sign, one bit of the word will be inhabited by the sign, and the magnitude will be composed of n -1 bits.
- **The condition of divide-overflow** is checked by subtracting the divisor in B from the half of bits of the dividend stored in A.
- **If A ≥ B, DVF is set and the operation is terminated before time.**
- **If A < B, no overflow condition occurs and so the value of the dividend is reinstated by adding B to A**.

Normal Division Process using Flowchart:

- The division of the magnitudes starts by shl dividend in AQ to left in the high-order bit shifted into E.
- Note – If shifted a bit into E is equal to 1, and we know that EA > B as EA comprises a 1 followed by n -1 bits whereas B comprises only n -1 bits). In this case, B must be subtracted from EA, and 1 should insert into Q, for the quotient bit.
- If the shift-left operation (shl) inserts a 0 into E, the divisor is subtracted by adding its 2's complement value and the carry is moved into E.
- If E = 1, it means that A ≥ B; thus, Q, is set to 1. If E = 0, it means that A < B and the original number is restored or reimposed by adding B into A.
- Now, this process is repeated with register A containing the partial remainder. After n-1 times, the final result is available in A and Q registers.

Example of Binary Division with Digital Hardware:



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.

2.7.1. 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 x re

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 x 103

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 + (247 – 1), which is approximately + 1014. 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) x 22047

This number is derived from a fraction that contains 35 1’s, an exponent of 11 bits(excluding its sign), and because 211–1 = 2047. The largest number that can be accommodated is approximately 10615. 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
(235 –1). This is approximately equal to 1010, 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 x 102

+ .1580000 x 10-1

It is necessary to make two exponents be equal before the mantissas can be added. We can either shift the first number three positions to the left, or shift the second number three positions to the right. When we store the mantissas in registers, shifting to the left causes a loss of most significant digits. Shifting to the right causes a loss of least significant digits. The second method is preferable because it only reduces the accuracy, while the first method may cause an error. The usual alignment procedure is to shift the mantissa that has the smaller exponent to the right by a number of places equal to the difference between the exponents. Now, the mantissas can be added.

. 5372400 x 102

+. 0001580 x 102

. 5373980 x 102

When two normalized mantissas are added, the sum may contain an overflow digit. An overflow can be corrected easily by shifting the sum once to the right and incrementing the exponent. When two numbers are subtracted, the result may contain most significant zeros as shown in the following example:

.56780 x 105

- .56430 x 105

.00350 x 105

An underflow occurs if a floating-point number that has a 0 in the most significant position of the mantissa. To normalize a number that contains an underflow, we shift the mantissa to the left and decrement the exponent until a nonzero digit appears in the first position. Here, it is necessary to shift left twice to obtain .35000 x 103. In most computers a normalization procedure is performed after each operation to ensure that all results are in a normalized form. 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.

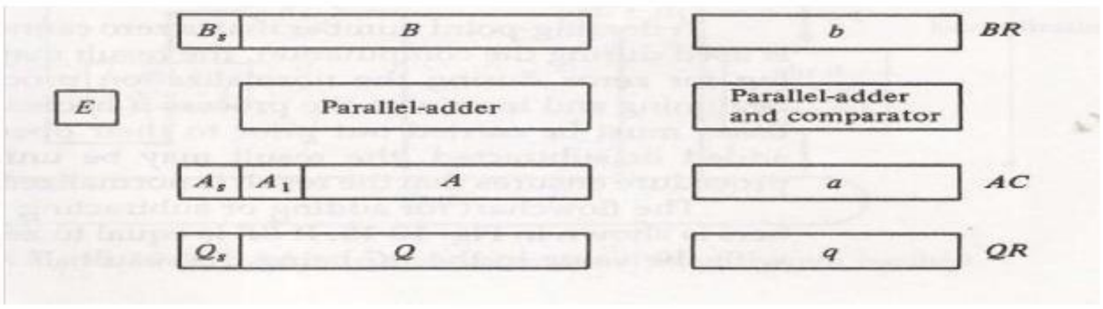
The operations done with the mantissas are the same as in fixed-point numbers, so the two can share the same registers and circuits.

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. A is a fourth representation also, known as a biased exponent.

In this representation, the sign bit is removed from beginning to form a separate entity. The bias is a positive number that is added to each exponent as the floating-point number is formed, so that internally all exponents are positive. The following example may clarify this type of representation. Consider an exponent that ranges from –50 to 49. Internally, it is represented by two digits (without a sign) by adding to it a bias of 50. The exponent register contains the number e + 50, where e is the actual exponent. This way, the exponents are represented in registers as positive numbers in the range of 00 to 99. Positive exponents in registers have the range of numbers from 99 to 50. The subtraction of 50 gives the positive values from 49 to 0. Negative exponents are represented in registers in the range of –1 to –50. Biased exponents have the advantage that they contain only positive numbers. Now it becomes simpler to compare their relative magnitude without bothering about their signs. Another advantage is that the smallest possible biased exponent contains all zeros. The floating-point representation of zero is then a zero mantissa and the smallest possible exponent.

<u>2.7.2.</u> <u>Register Configuration</u>

The register configuration for floating-point operations is shown in figure 4.13. 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 Fig. 4.13. 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.



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 is 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 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.

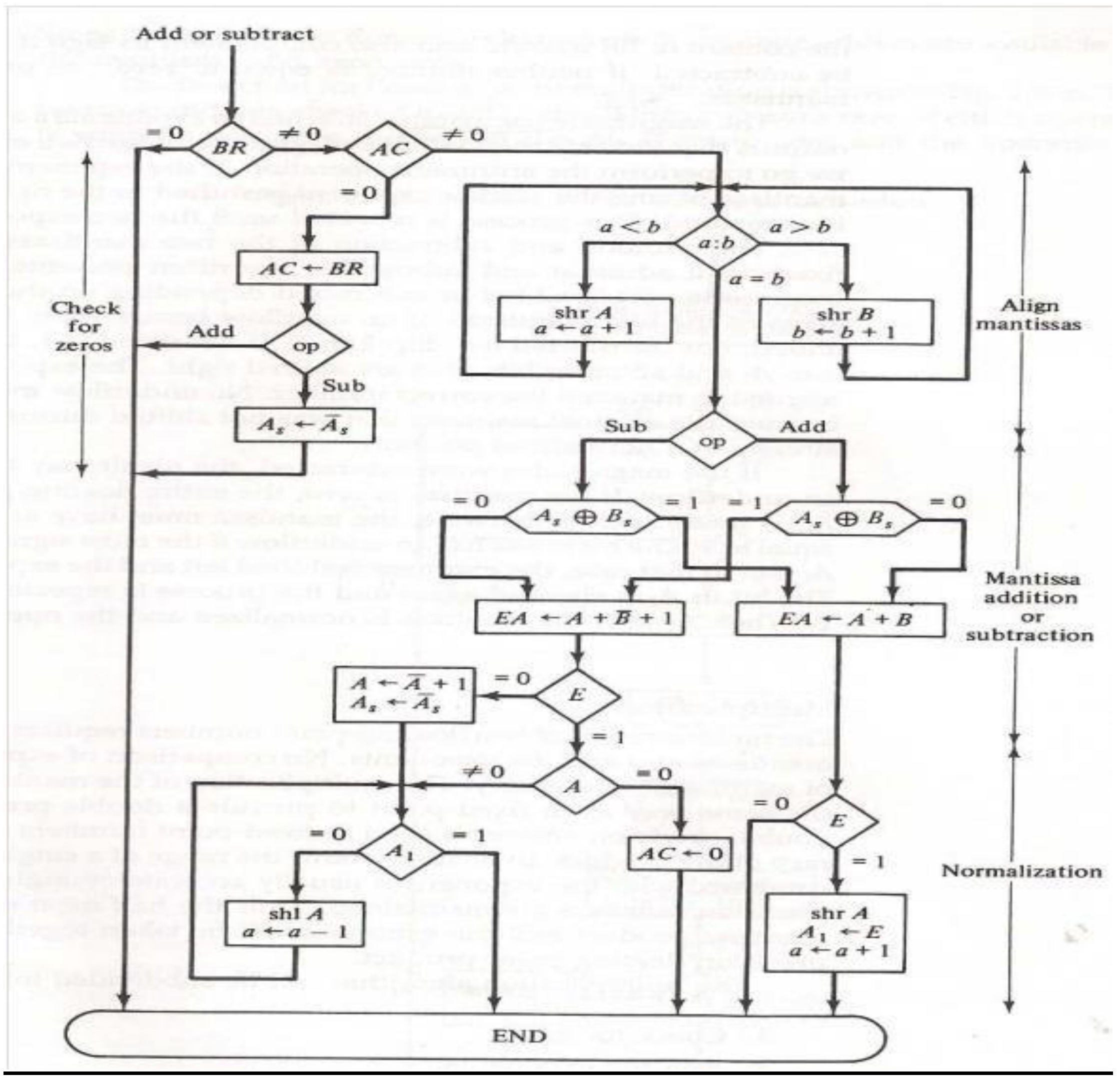
2.7.3. Addition and Subtraction of Floating Point Numbers

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:

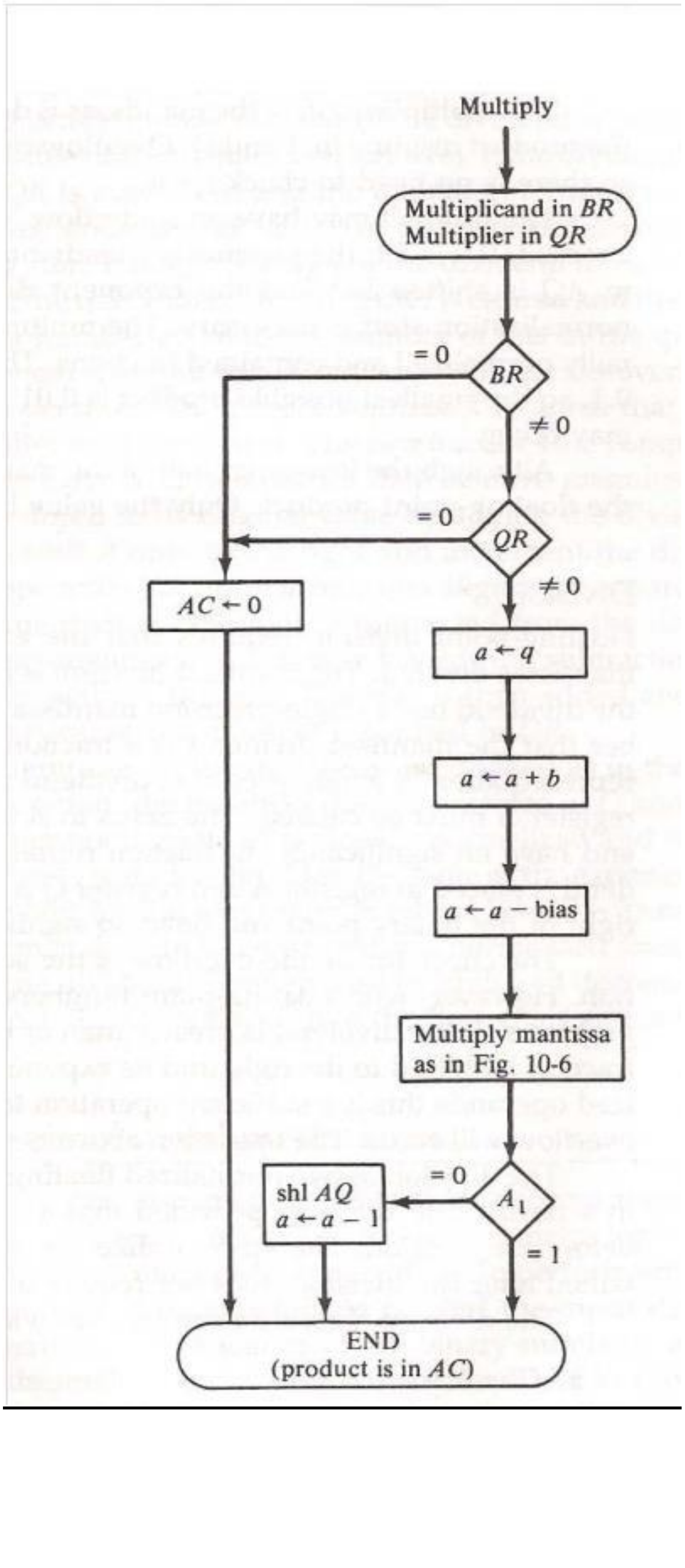
1. Check for zeros.
2. Align the mantissas.
3. Add or subtract the mantissas
4. Normalize the result

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. For adding or subtracting two floating-point binary numbers, if BR is equal to zero, the operation is stopped, with the value in the AC being the result. If AC = 0, we transfer the content of BR into AC and also complement its sign we have to subtract the numbers. If neither number is equal it to zero, we proceed to align the mantissas. The magnitude comparator attached to exponents a and b gives three outputs, which show their relative magnitudes. If the two exponents are equal, we go to perform the arithmetic operation. If the exponents are not equal, the mantissa having the smaller exponent is shifted to the right and its exponent incremented. This process is repeated until two exponents are equal.

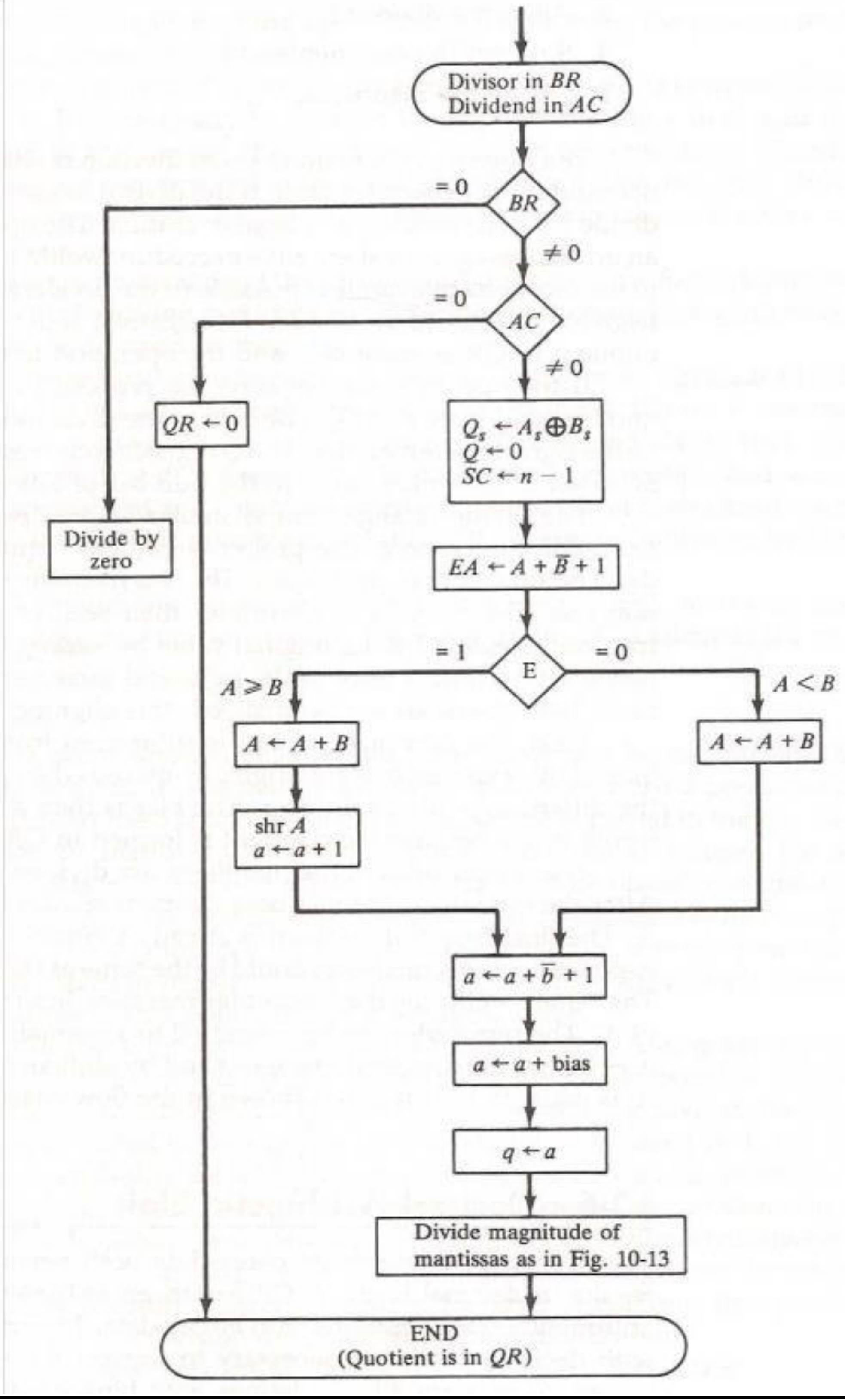
The addition and subtraction of the two mantissas is similar to the fixed-point addition and subtraction algorithm presented in Fig. 4.14. The magnitude part is added or subtracted depends on the operation and the signs of the two mantissas. If an overflow occurs when the magnitudes are added, it is transferred into flip-flop E. If E = 1, the bit is transferred into A1 and all other bits of A are shifted right. The exponent must be incremented so that it can maintain the correct number. No underflow may occur in this case this is because the original mantissa that was not shifted during the alignment was already in a normalized position. 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.



2.7.4. Multiplication



2.7.5. Division



2.8. DECIMAL ARITHMETIC OPERATIONS:

2.8.1. Decimal Arithmetic Unit:

The user of a computer input data in decimal numbers and receives output in decimal form. But a CPU with an ALU can perform arithmetic micro-operations only on 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 that can do decimal arithmetic must store the decimal data in binary coded form. The decimal numbers are then applied to a decimal arithmetic unit, which can execute decimal arithmetic micro-operations.

Electronic calculators invariably use an internal decimal arithmetic unit since inputs and outputs are frequent. There does not seem to be a reason for converting the keyboard input numbers to binary and again converting the displayed results to decimal, this is because this process needs special circuits and also takes a longer time to execute. Many computers have hardware for arithmetic calculations with both binary and decimal data.

Users can specify by programmed instructions whether they want the computer to does calculations with binary or decimal data. A decimal arithmetic unit is a digital function that does decimal micro-operations. It can add or subtract decimal numbers. The unit needs coded decimal numbers and produces results in the same adopted binary code. A single- stage decimal arithmetic unit has of nine binary input variables and five binary output variables, since a minimum of four bits is required to represent each coded decimal digit. Each stage must have four inputs for the addend digit, four inputs for the addend digit, and an input-carry. The outputs need four terminals for the sum digit and one for the output carry. Of course, there is a wide range of possible circuit configurations dependent on the code used to represent the decimal digits.

2.8.2. BCD Adder:

Now let us see the arithmetic addition of two decimal digits in BCD, 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. Assume 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 4.4 and are labeled by symbols K, Z8, Z4, Z2, and Z1. 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 its 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 column of the table. The problem is to find a simple rule so that the binary number in the first column can be converted to the correct BCD digit representation of the number in the second column. It is apparent that when the binary sum is equal to or less than 1001, no conversion is needed. When the binary sum is greater than 1001, we need to add of binary 6 (0110) to the binary sum to find the correct BCD representation and to produces output-carry as required.

One way of adding decimal numbers in BCD is to use 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. The second operation produces 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

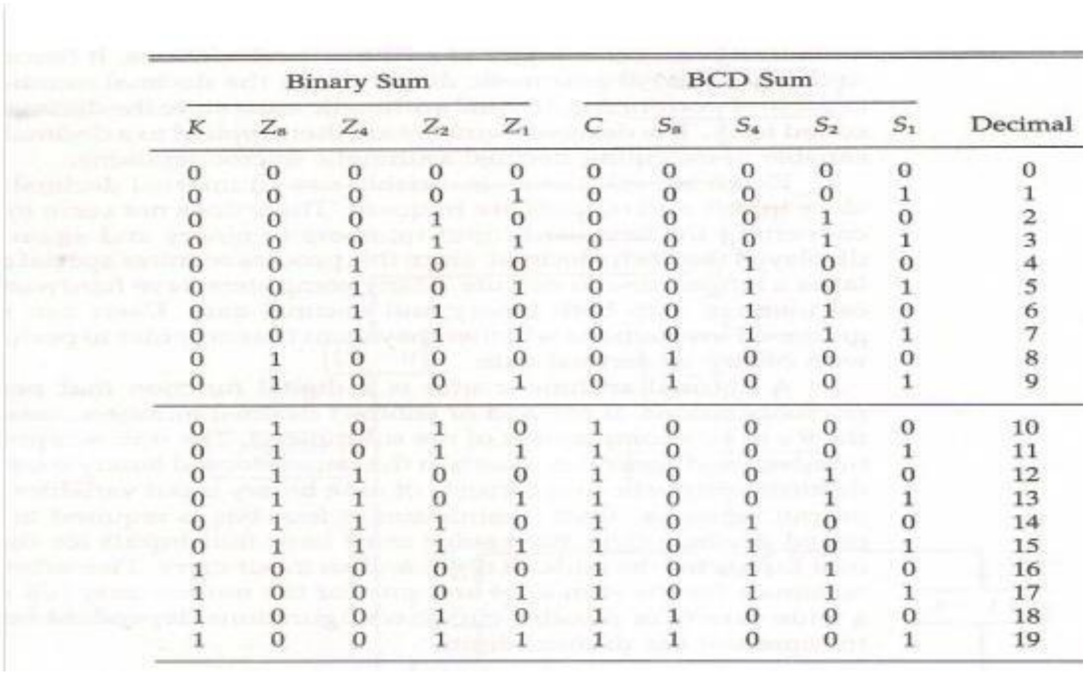
all decimal digits are added.

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

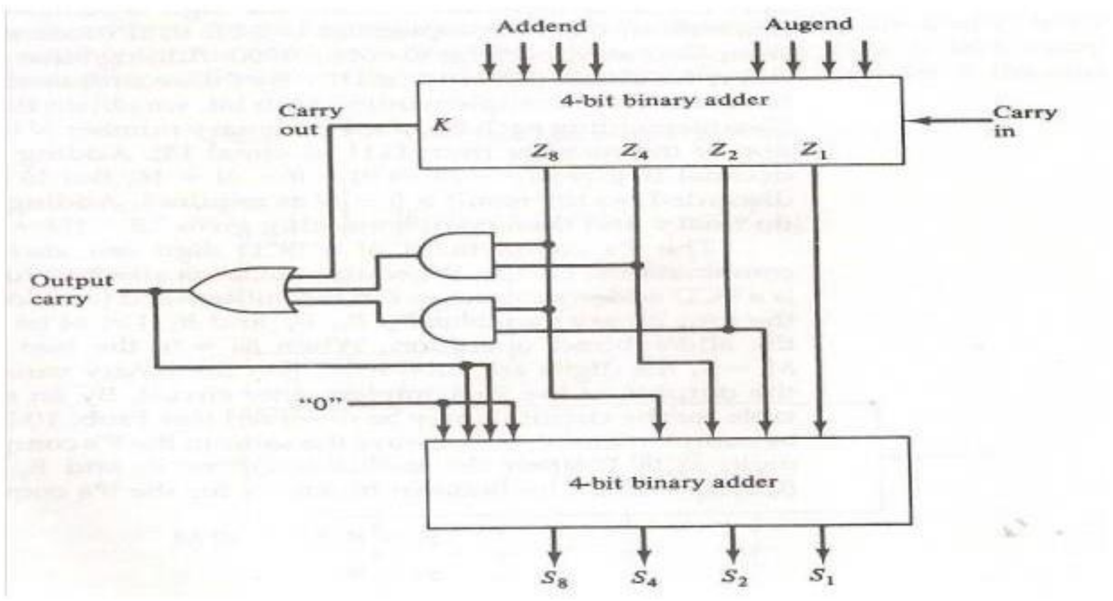
1010 to 1111 that need a correction have a 1 in position Z8. To differentiate them from binary 1000 and 1001, which also have a 1 in position Z8, we specify further that either Z4 or Z2 must have a 1. The condition for a correction and an output-carry can be expressed by the Boolean function

$$C = K + Z8\,Z4 + Z8\,Z2$$

When C = 1, we need to add 0110 to the binary sum and provide an output-carry for the next stage. A BCD adder is circuit that adds two BCD digits in parallel and generates a sum digit also in BCD. ABCD 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. 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 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.



Binary Sum					BCD Sum					
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	Decimal
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



2.7.1. BCD Subtraction:

Subtraction of two decimal numbers needs a subtractor circuit that is different from a BCD adder. We 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, we cannot obtain the 9's complement by complementing each bit in the code. It must be formed using 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 but we have to include. There are two possible correction methods. In the first method, binary 1010 (decimal 10) is added to each complemented digit then we discard the carry after each addition.

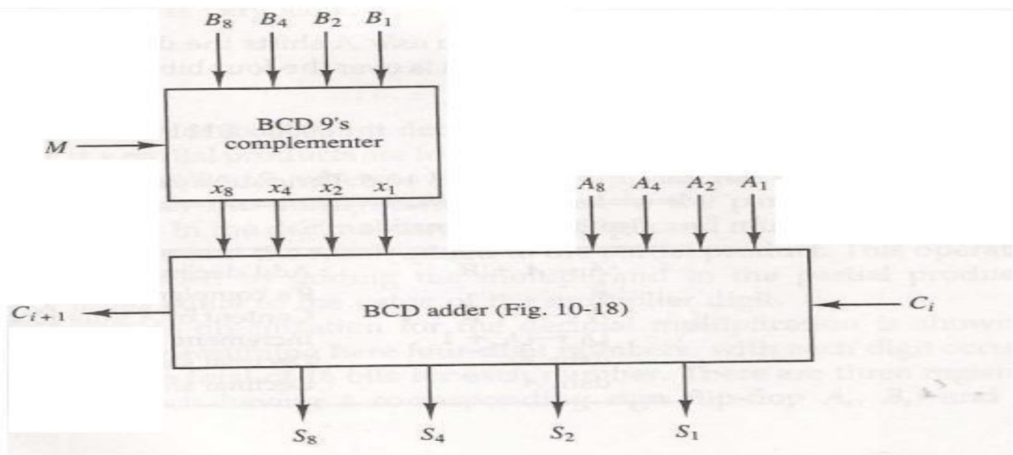
In the second method, binary 0110 (decimal 6) is added before the digit is complemented. As a numerical illustration, the 9's complement of BCD 0111(decimal 7) is computed by first complementing each bit to obtain 1000. Adding binary 1010 and discarding the carry, we obtain 0010 (decimal 2). By the second method, we add 0110
to 0111 to obtain 1101. Complementing each bit, we obtain the required result of 0010. Complementing each bit of 4-bit binary number N is identical to the subtraction of the number from 1111 (decimal 15). Adding

the binary equivalent of decimal 10 gives 15 – N + 10 = 9 + 16. But 16 signifies the carry that is discarded, so the result is 9 – N as required. Adding the binary equivalent of decimal 6 and then complementing gives 15 – (N + 6) = 9 – N as required.

We can also obtain the 9's complement of a BCD digit through a combinational circuit. When this circuit is combined to a BCD adder, we get a BCD adder/subtractor. Let the subtrahend (or addend) digit be denoted by the four binary variables B8, B4, B2, and B1. 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 x8, x4, x2, and x1 be the outputs of the 9's complement circuit. By an examination of the truth table for the circuit, it may be observed that B1 should always be complemented; B2 is always the same in the 9's complement as in the original digit; x4 is 1 when the exclusive OR of B2 and B4 is 1; and x8 is 1 when B8B4B2 = 000. The Boolean functions for the 9's complement circuit are

x1 = B1 M' + B'1 M x2 = B2
x4 = B4M' + (B'4B2 + B4B'2)M x8 =
B8M' + B'8B4'B'2M

From these equations we see that x = B when M = 0. When M = 1, the x equals to the 9's complement of B. One stage of a decimal arithmetic unit that can be used to add or subtract two BCD digits is given in Fig.
4.18. It has 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 the 9's complement of
B. For numbers with n decimal digits we need n such stages. The output carries Ci+1 from one stage. to subtract the two decimal numbers let M = 1 and apply a 1 to the input carry C1 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.



B_8 B_4 B_2 B_1
BCD 9's complementer
M
x_8 x_4 x_2 x_1
A_8 A_4 A_2 A_1
C_{i+1}
BCD adder (Fig. 10-18)
C_i
S_8 S_4 S_2 S_1