UNIT – IV
COMPUTER ARITHMETIC

Objective:

- To familiarize with the algorithms to perform arithmetic operations.

Syllabus:

Data representation- fixed point, floating point, addition and subtraction, multiplication and division algorithms.

Learning Outcomes:

At the end of the unit student will be able to:

- Represent data in fixed and floating point formats.
- Perform arithmetic operations on data represented in fixed point, floating point formats.

Learning Material

4.1 DATA REPRESENTATION

- For positive numbers sign bit equal to 0 and 1 for negative numbers.
- In addition to the sign, a number may have a binary (or decimal) point.
- The position of the binary point is needed to represent fractions, integers, or mixed integer-fraction numbers.
- There are two ways of specifying the position of the binary point in a register: by giving it a fixed position or by employing a floating-point representation.
- The fixed-point method assumes that the binary point is 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.
    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, but its presence is assumed from the fact that the number stored in the register is treated as a fraction or as an integer.

4.1.1 Integer Representation

- When an integer binary number is positive, the sign is represented by 0 and the magnitude by a positive binary number.
- When the number is negative, the sign is represented by 1 but the rest of the number may be represented in one of three possible ways:

1. Signed Magnitude representation
2. Signed 1's complement representation
3. Signed 2's complement representation

Eg: There is only one way to represent + 14, there are three different ways to represent -14 with eight bits.

In signed-magnitude representation	0 0001110	+14
In signed-magnitude representation	1 0001110	-14
In signed-1's complement representation	1 1110001	-14
In signed-2's complement representation	1 1110010	-14

4.1.2 Arithmetic Addition

Signed-magnitude Addition

- The addition of two numbers in the ***signed-magnitude*** system follows the rules of ordinary arithmetic.
- If the signs are the same, we add the two magnitudes and give the sum the common sign.
- If the signs are different, we subtract the smaller magnitude from the larger and give the result the sign of the larger magnitude.
- For example, (+25) + (-37) = -(37 - 25) = -12 and is done by subtracting the smaller magnitude 25 from the larger magnitude 37 and using the sign of 37 for the sign of the result.

Signed 2's complement addition

- The addition of two numbers in the signed 2's complement addition system follows.
- Add the two numbers, including their sign bits, and discard any carry out of the sign (leftmost) bit position. Numerical examples for addition are shown below.

Eg 1:

```
+6      00000110
+13     00001101
+19     00010011
```

Eg 2:

```
-6      11111010          (Signed 2's complement of -6)
+13     00001101
+7    1]00000111
```

Eg3:

```
+6      00000110
-13     11110011          (Signed 2's complement of -13)
-7      11111001          (Signed 2's complement of -7)
```

Eg4:

-6	11111010	(Signed 2's complement of -6)
-13	11110011	(Signed 2's complement of -13)
-19	1] 11101101	(Signed 2's complement of -19)

4.1.3 Arithmetic Subtraction

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

Eg 1:

-6	11111010	(Signed 2's complement of -6)
-13	11110011	(Signed 2's complement of -13)

-6	11111010	
-13	00001101	(2's complement of -13)
+7	1] 00000111	

Eg 2:

+13	00001101
+6	00000110

+13	00001101	
+6	11111010	(2's complement of +6)
+7	1] 00000111	

Eg 3:

-6	11111010	(Signed 2's complement of -6)
+13	00001101	

-6	11111010	
+13	11110011	(2's complement of +13)
-19	1] 11101101	

Eg 4:

-13	11110011	(Signed 2's complement of -13)
-6	11111010	(Signed 2's complement of -6)

-13	11110011	
-6	00000110	(2's complement of -6)
-7	11111001	(Signed 2's complement of -7)

** When ever Addition or Subtraction operation is performed on two numbers which are represented using signed 2's complement form, if result is –ve number, i.e. MSB is 1, then that indicates result is also in Signed 2's complement form.

4.1.4 Overflow

- When two numbers of n digits each are added and the sum occupies n + 1 digits, we say that an overflow occurred.
- An overflow is a problem in digital computers because the width of registers is finite and fixed. A result that contains n + 1 bits cannot be accommodated in a register with a standard length of n bits. For this reason, many computers detect the occurrence of an overflow, and when it occurs, a corresponding flip-flop is set which can then be checked by the user.
- An overflow cannot occur after an addition if one number is positive and the other is negative.

```
carries: 0 1                 carries: 1 0
     +70   0 1000110              −70   1 0111010
     +80   0 1010000              −80   1 0110000
    +150   1 0010110             −150   0 1101010
```

- An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position.
- If these two carries are *not equal*, an overflow *condition is produced*.
- If these two carries are *equal*, an overflow *condition is not produced*.

4.2 ADDITION AND SUBTRACTION

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

4.2.1 Addition and Subtraction with Signed-Magnitude Data

- The algorithms for addition and subtraction stated as follows (the words inside parentheses should be used for the *subtraction* algorithm):
- Addition (subtraction) algorithm.

  1. when the signs of A and B are identical (*different*), add the two magnitudes and attach the sign of A to the result.

  2. When the signs of A and B are different (*identical*), compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to be the same as A if A > B or the complement of the sign of A if A < B. If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

Table 1: Addition and Subtraction of Signed Magnitude Numbers

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

4.2.2 Hardware Implementation for Addition and Subtraction with Signed-Magnitude Data

- To implement the two arithmetic operations with hardware, it is first necessary that the two numbers be stored in registers.
- Let A and B be two registers that hold the magnitudes of the numbers, and A_s, and B_s be two flip-flops that hold the corresponding signs.
- Here parallel-adder is needed to perform the microoperation A + B. (Consists of Full adder).
- The complementer for generating the 2's Complement while performing subtraction operation. (Consists of X-OR gate).

- Where M is Mode of operation. When M = 0, the output of B is transferred to the adder, the input carry is 0, and the adder is equal to the sum A + B.
- When M =1, the l's complement of B is applied to the adder, the input carry is 1, and output is = $A + \bar{B} + 1$. This is equal to A plus the 2's complement of B, which is equivalent to the A – B.

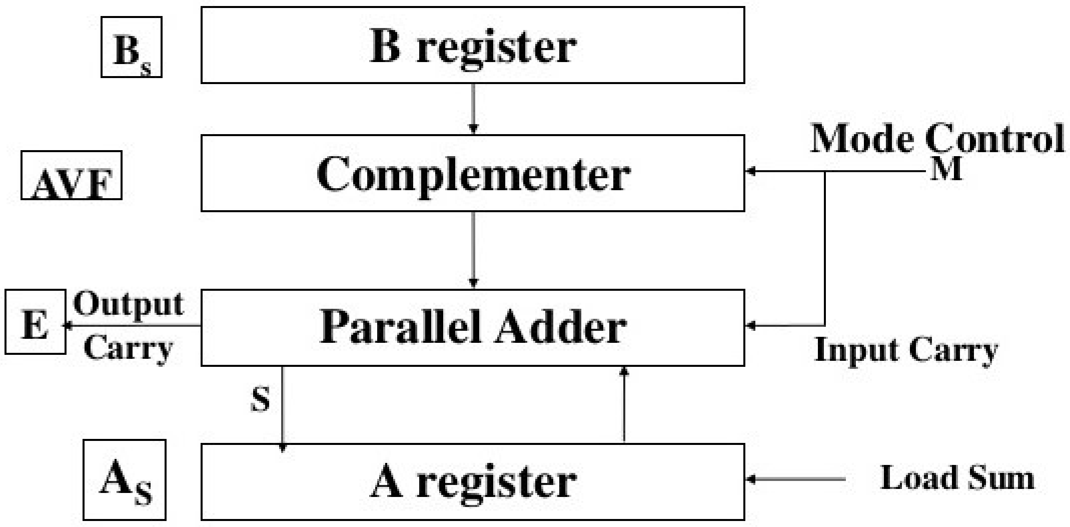


Figure 1: Hardware for signed-magnitude addition and subtraction

4.2.3 Hardware Algorithm for Signed-magnitude addition and subtraction

- The magnitudes are added with a microoperation EA←A + B. where EA is a register that combines E and A.
- The value of E is transferred into the Add-overflow flip-flop AVF, if E is 1.
- The magnitudes are subtracted by adding A to the 2's complement of B.
- No overflow can occur if the numbers are subtracted so AVF is cleared to 0.
- 1 in E indicates that A≥ B and the number in A is the correct result. If this number i.s 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. This operation can be done with one mlcrooperation $A \leftarrow \bar{A} + 1$.

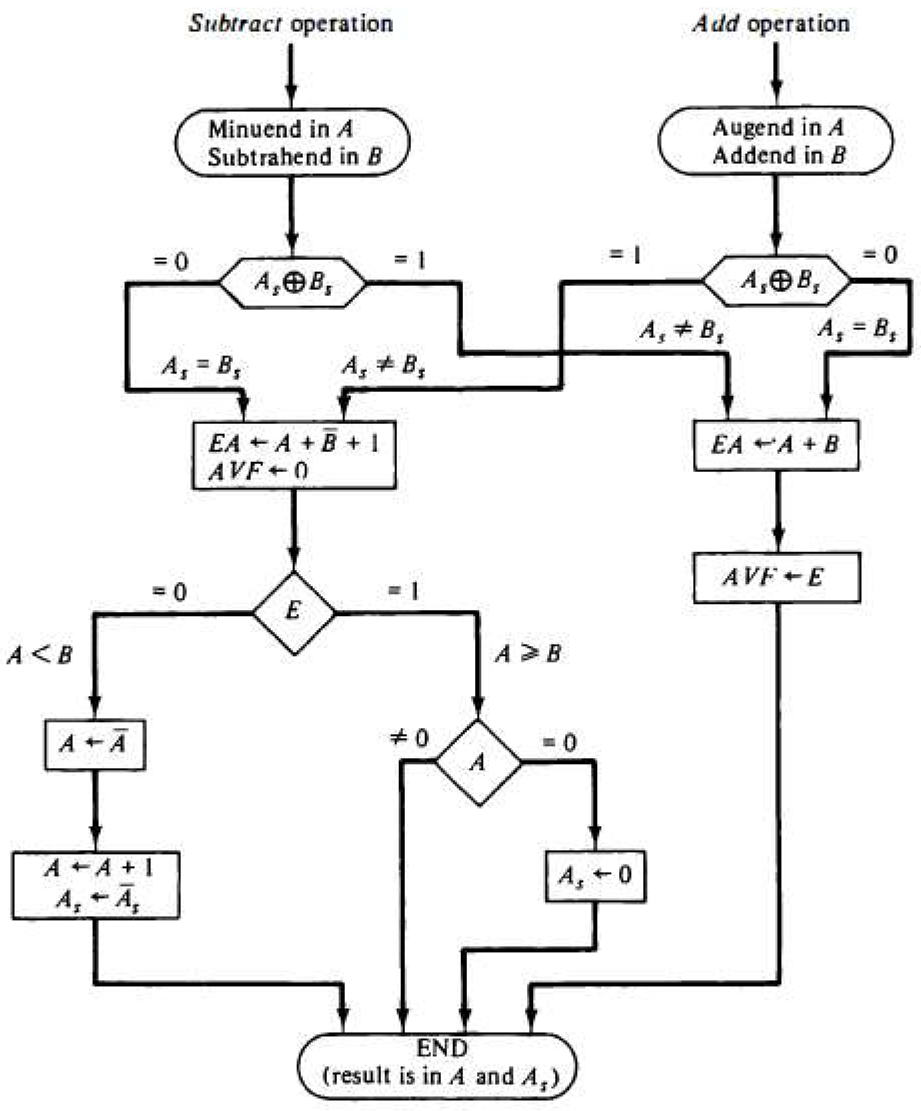


Figure 2: Flowchart for add and subtract operations

4.2.4 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 3.

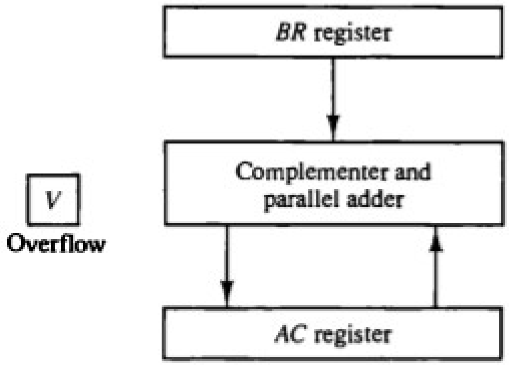


Figure 3: Hardware for signed2's complement addition and subtraction

- The algorithm for adding and subtracting two binary numbers in signed2's complement representation is shown in the flowchart of Figure 4.

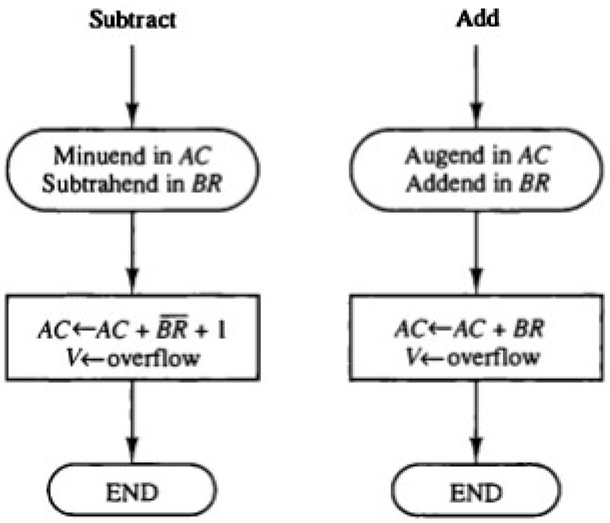


Figure 4: Algorithm for adding and subtracting numbers in signed 2's complement representation

- 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.

4.3 MULTIPLICATION ALGORITHMS

- Multiplication of two fixed-point binary numbers in signed-magnitude representation is done with process of successive shift and add 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
```

4.3.1 Hardware Implementation for Signed-Magnitude Data Multiplication

- The hardware for multiplication consists of the equipment shown in Figure 5.

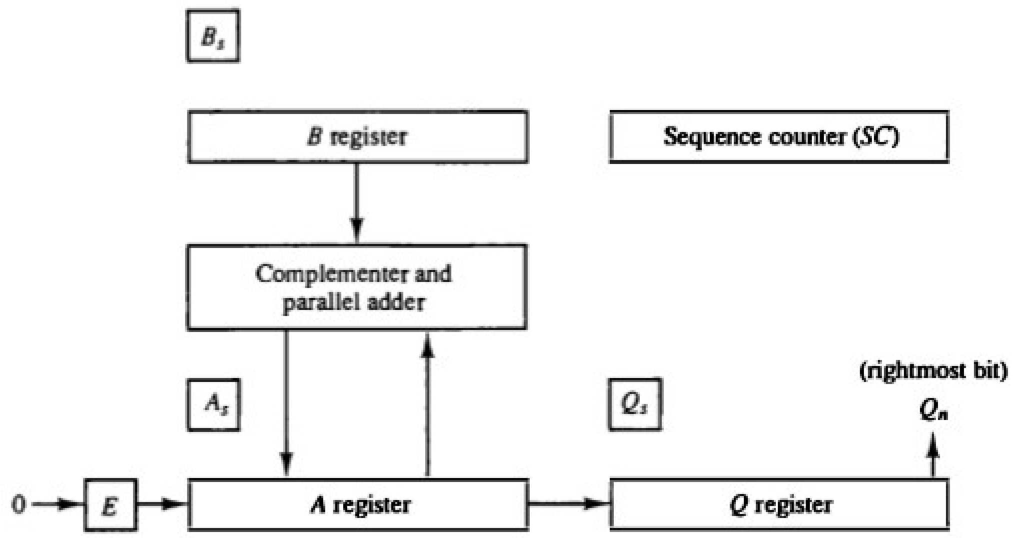


Figure 5: Hardware for multiply operation

- Initially, the multiplicand is in register B and the multiplier in Q.
- 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 5.

- 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.

4.3.2 Hardware Algorithm for Signed-Magnitude Data Multiplication

- The following figure 6 is a flowchart of the hardware multiply algorithm.

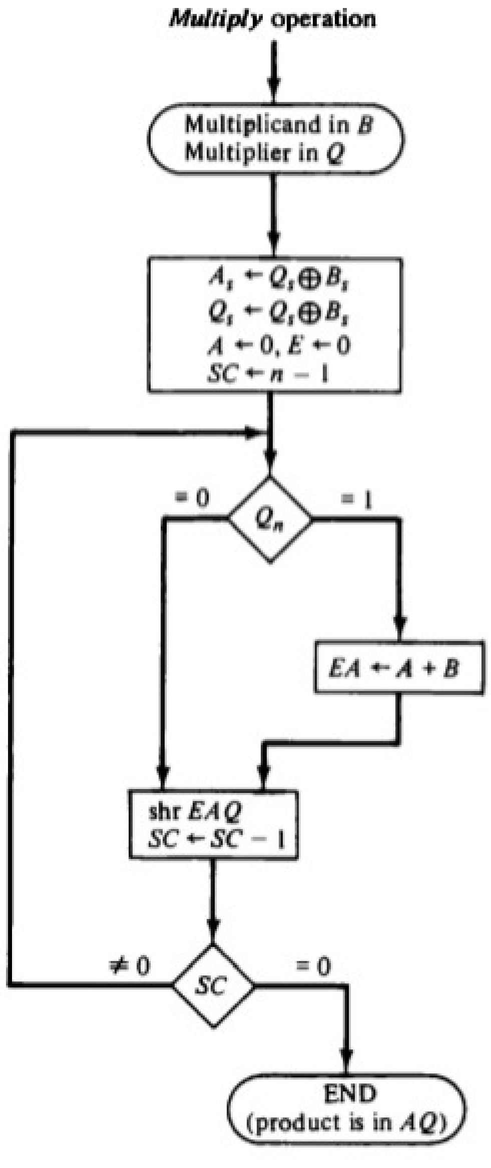
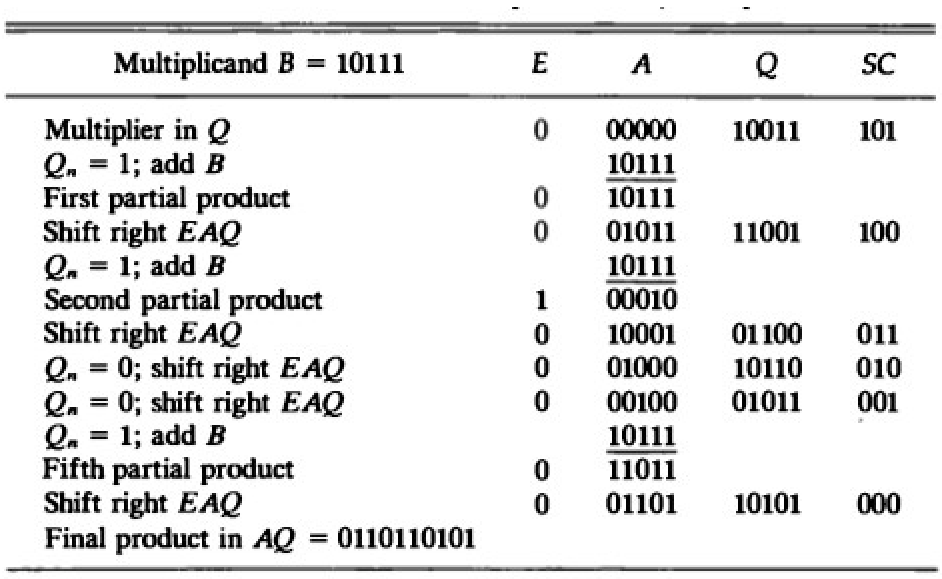


Figure 6: Flowchart multiply operation on sign magnitude representation numbers

- Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in B_s and Q_s, respectively.
- The signs are compared, and both 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 Q, is tested. If it is a 1, the multiplicand in B is added to the present partial product in A. If it is a 0, nothing is done.
- 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 2: 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 $A_s = B_s \oplus Q_s$, i.e $A_s = 0 \oplus 0 \Rightarrow 0$.

Table 3: Numerical Example for Binary Multiplier

Multiplicand B=10011	**E**	**A**	**Q**	**SC**
Initially	0	00000	00110	5
$Q_n = 0$. So shr EAQ	0	00000	00011	4
$Q_n = 1$. So add B to A		00000 $\underline{10011}$ 10011		
Now shr EAQ	0	01001	10001	3
$Q_n = 1$. So add B to A		01001 $\underline{10011}$ 11100		
Now shr EAQ	0	01110	01000	2
$Q_n = 0$. So shr EAQ	0	00111	00100	1
$Q_n = 0$. So shr EAQ	0	**00011**	**10010**	0

Now Result is available in Registers A and Q. i.e. 0001110010 => 114 and sign bit of A is 0. So result is +114.

- The following table 4 describes multiplication of binary numbers **10010(-18)** and **00110(+5)** which are represented using Sign Magnitude Representation.
- Here Multiplicand is negative value, so $B_s = 1$. Here Multiplier is positive value, so $Q_s = 0$.
- Now $A_s = B_s \oplus Q_s$,i.e $A_s = 1 \oplus 0 \Rightarrow 1$.

Table 4: Numerical Example for Binary Multiplier

Multiplicand B=10010	**E**	**A**	**Q**	**SC**
Initially	0	00000	00101	5
Q_n = 1. So add B to A		00000 $\underline{10010}$ 10010		
Now shr EAQ	0	01001	00010	4
Q_n = 0. So shr EAQ	0	00100	10001	3
Q_n = 1. So add B to A		00100 $\underline{10010}$ 10110		
Now shr EAQ	0	01011	01000	2
Q_n = 0. So shr EAQ	0	00101	10100	1
Q_n = 0. So shr EAQ	0	**00010**	**11010**	0

Now Result is available in Registers A and Q. i.e. 0001011010 => 90 and sign bit of A is 1.
So result is **-90**.

4.3.3 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 7.

- 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.

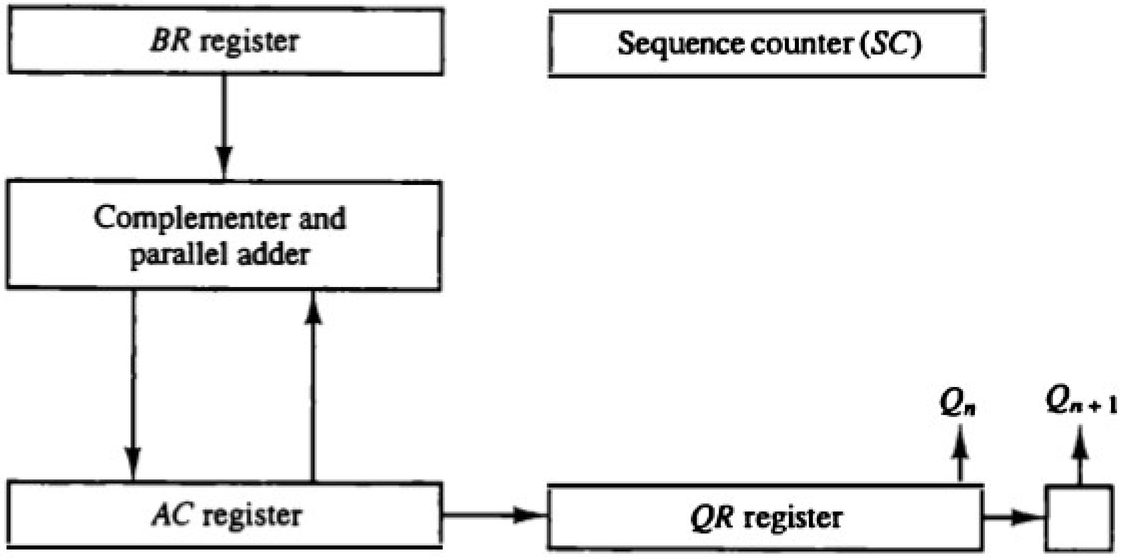


Figure 7: Hardware for Booth algorithm

- The flowchart for Booth algorithm is shown in Figure 8.
- 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 the partial 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.

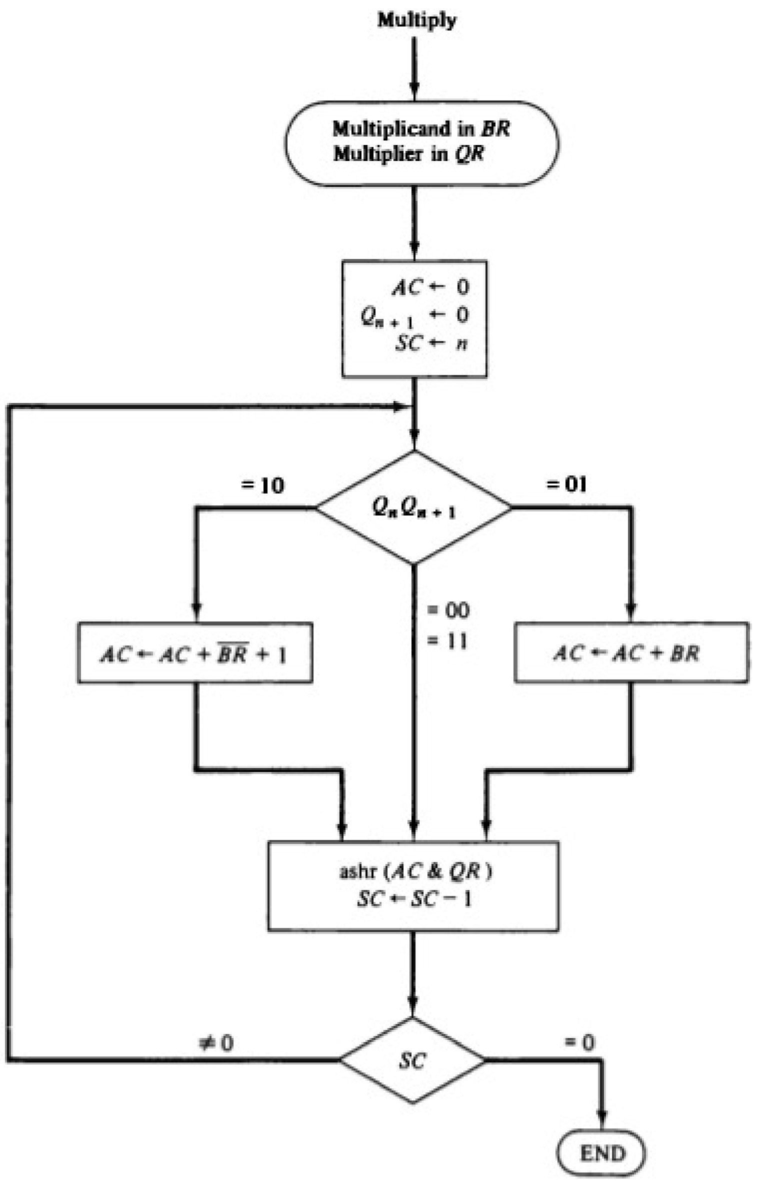


Figure 8: Booth algorithm for multiplication of signed 2's complement numbers.

- 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.

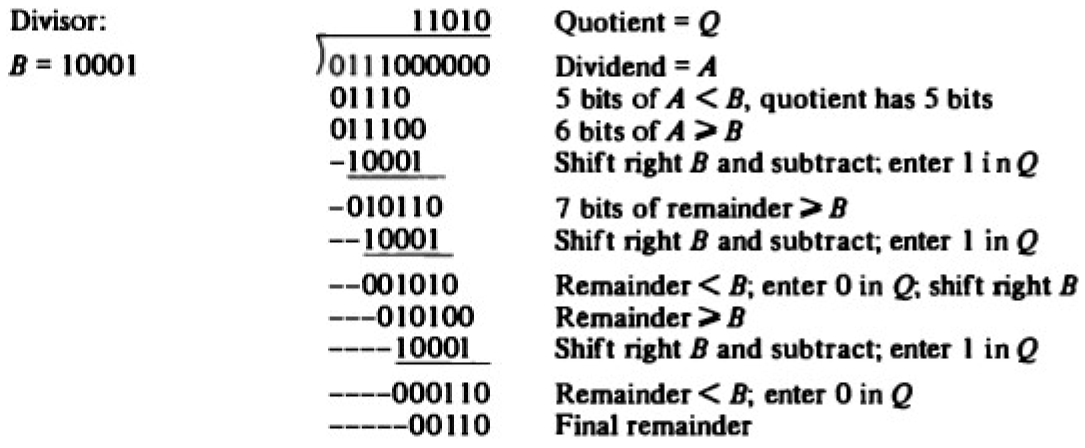
Table 5: Example of Multiplication with Booth Algorithm

$Q_n\, Q_{n+1}$	$BR = 10111$ $\overline{BR} + 1 = 01001$	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

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

4.4 DIVISION ALGORITHMS

- The division process is illustrated by a numerical example in Figure 9. The divisor B consists of five bits and the dividend A, of ten bits.



```
Divisor:            11010        Quotient = Q
B = 10001     )0111000000        Dividend = A
               01110             5 bits of A < B, quotient has 5 bits
               011100            6 bits of A ≥ B
               -10001            Shift right B and subtract; enter 1 in Q
               -010110           7 bits of remainder ≥ B
               --10001           Shift right B and subtract; enter 1 in Q
               --001010          Remainder < B; enter 0 in Q; shift right B
               ---010100         Remainder ≥ B
               ----10001         Shift right B and subtract; enter 1 in Q
               ----000110        Remainder < B; enter 0 in Q
               -----00110        Final remainder
```

Figure 9: Example of binary division

- The five most significant bits of the dividend are compared with the divisor. Since the 5-bit number is smaller than B, we try again by taking the six most significant bits of A and compare this number with B.

- The 6-bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend. The divisor is then shifted once to the right and subtracted from the dividend. The difference is called a partial remainder.
- The process is continued by comparing a partial remainder with the divisor. If the partial remainder is greater than or equal to the divisor, the quotient bit equal to 1. The divisor is then shifted right and subtracted from the partial remainder.
- If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction is needed. The divisor is shifted once to the right in any case. Note that the result gives both a quotient and a remainder.

4.4.1 Hardware Implementation for Signed-Magnitude Data Division operation

- When the division is implemented in a digital computer, it is convenient to change the process slightly. Instead of shifting the divisor to the right, the dividend, or partial remainder, is shifted to the left.
- Subtraction may be achieved by adding A to the 2's complement of B.
- The hardware for implementing the division operation is identical to that required for multiplication. Register EAQ is now *shifted to the left* with 0 inserted into Q, and the previous value of E lost.

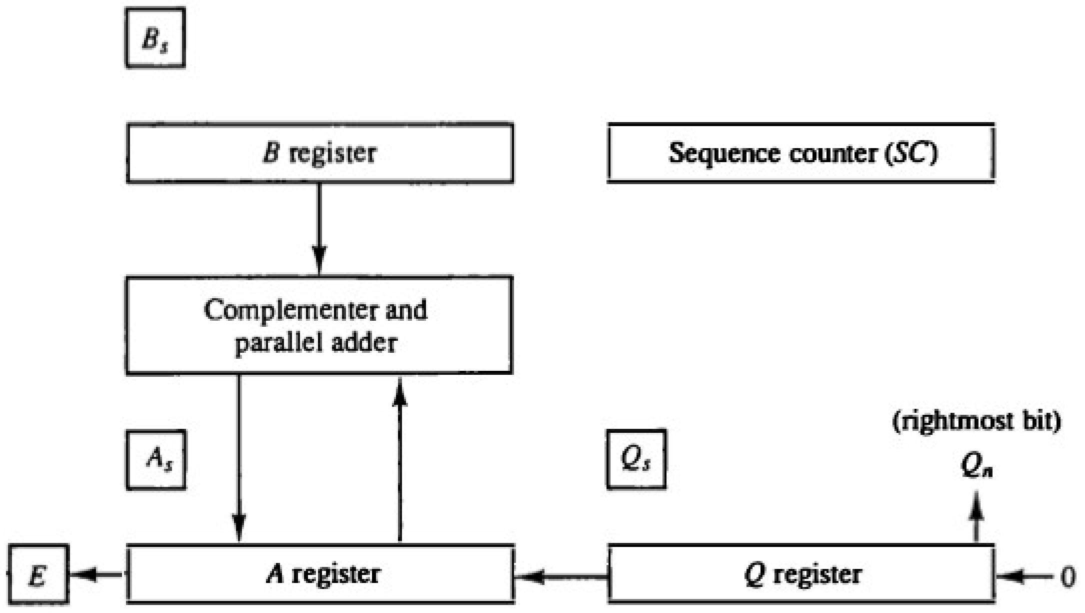
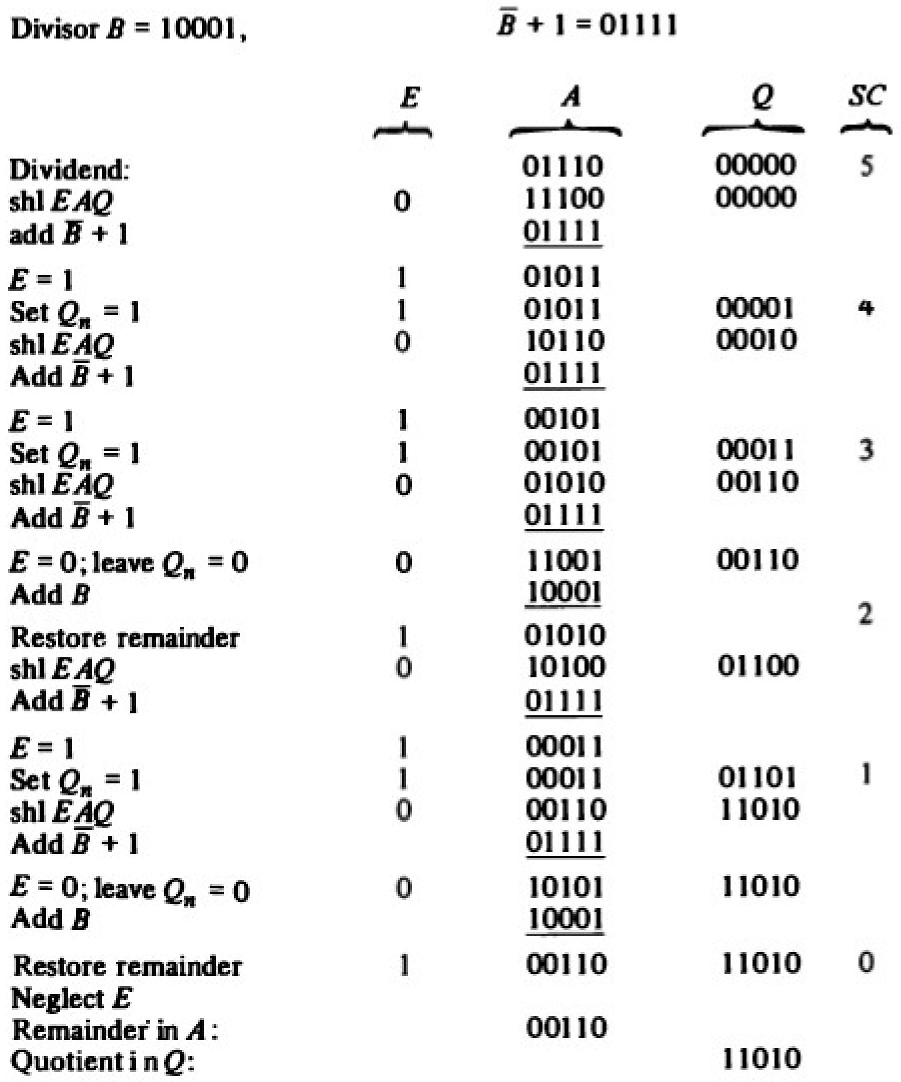


Figure 10: Hardware for division operation



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

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

Figure 11: Example of binary division with digital hardware

- The divisor is stored in register B and the double-length dividend is stored in registers A and Q.
- The dividend is shifted to the left and the divisor is subtracted by adding its 2's complement value. The information about the relative magnitude is available in E.
- If E = 1, it signifies that A ≥ B. 1 is inserted into Q_n(LSB of Quotient), and the partial remainder is shifted to the left to repeat the process.
- If E = 0, it signifies that A < B so the quotient in Q_n, remains 0 (inserted during the shift). The value of B is then added to restore the partial remainder in A to its previous value.
- The partial remainder is shifted to the left and the process is repeated again until the partial remainder obtained is smaller than the divisor.
- Finally, the quotient is in Q and the final remainder is in A.

- The sign of the quotient is determined from the signs of the dividend and the divisor. If the two signs are same, the sign of the quotient is plus. If they are different, the sign is minus. The sign of the remainder is the same as the sign of the dividend.

4.4.2 Divide Overflow

- The division operation may result in a quotient with an overflow.
- When the dividend is twice as long as the divisor, the condition for overflow can be stated as follows: A divide-overflow condition occurs if the high-order half bits of the dividend constitute a number greater than or equal to the divisor.
- Another problem associated with division is the fact that a division by zero must be avoided. This occurs because any dividend will be greater than or equal to a divisor which is equal to zero.
- Overflow condition is usually detected when a special flip-flop is set. We will call it a divide-overflow flip-flop and label it DVF.

4.4.3 Hardware Algorithm

- The hardware divide algorithm is shown in the flowchart of Figure 12. The dividend is in A and Q and the divisor in B. The sign of the result is transferred into Q_s, to be part of the quotient.
- A constant is set into the sequence counter SC to specify the number of bits in the quotient.
- A divide-overflow condition is tested by subtracting the divisor in B from half of the bits of the dividend stored in A.
- If $A \geq B$, the divide-overflow flip-flop DVF is set and the operation is terminated prematurely. If $A < B$, no divide overflow occurs so the value of the dividend is restored by adding B to A.
- The division of the magnitudes starts by shifting the dividend in AQ to the left with the high-order bit shifted into E. If the bit shifted into E is 1, we know that $EA > B$ because EA consists of a 1 followed by n-1 bits while B consists of only n-1 bits. In this case, B must be subtracted from EA and 1 inserted into Q_n, for the quotient bit.
- If the shift-left operation inserts a 0 into E, the divisor is subtracted by adding its 2's complement value and the carry is transferred into E.
- If $E = 1$, it signifies that $A \geq B$; therefore, Q, is set to 1.
- If $E = 0$, it signifies that $A < B$ and the original number is restored by adding B to A.
- This process is repeated again with register A holding the partial remainder. After n - 1 times, the quotient magnitude is formed in register Q and the remainder is found in register A.
- The quotient sign is in Q_s, and the sign of the remainder in A_s, is the same as the original sign of the dividend.

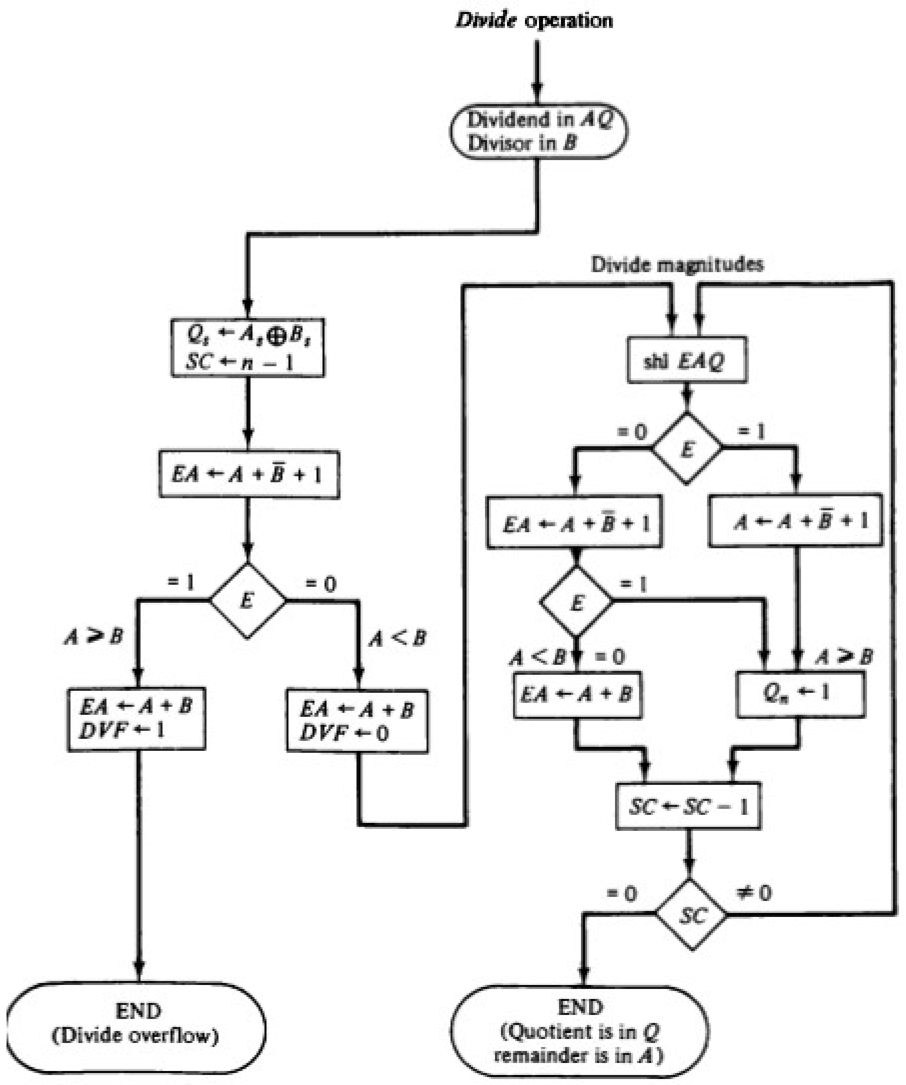


Figure 12: Flowchart for divide operation

4.4.4 Other Algorithms

- The hardware method just described is called the *restoring method*. The reason for this name is that the partial remainder is restored by adding the divisor to the negative difference.
- Two other methods are available for dividing numbers, the *comparison method* and the *non-restoring method*.
- In the *comparison method* A and B are compared prior to the subtraction operation. Then if $A \geq B$, B is subtracted from A. If $A < B$ nothing is done.
- The partial remainder is shifted left and the numbers are compared again. The comparison can be determined prior to the subtraction by inspecting the end-carry out of the parallel-adder prior to its transfer to register E.

- In the *non-restoring method*, B is not added if the difference is negative but instead, the negative difference is shifted left and then B is added. To see why this is possible consider the case when A < B. From the flowchart in Figure 12, we note that the operations performed are A - B + B; that is, B is subtracted and then added to restore A. The next time around the loop, this number is shifted left (or multiplied by 2) and B subtracted again. This gives 2(A - B + B) - B = 2A - B.
- This result is obtained in the non-restoring method by leaving A – B as it is. The next time around the loop, the number is shifted left and B added to give 2(A - B) + B = 2A - B, which is the same as before.
- Thus, in the non-restoring method, B is subtracted if the previous value of Q_n was a 1, but B is added if the previous value of Q_n was a 0 and no restoring of the partial remainder is required.
- This process saves the step of adding the divisor if A is less than B, but it requires special control logic to remember the previous result. The first time the dividend is shifted, B must be subtracted. Also, if the last bit of the quotient is 0, the partial remainder must be restored to obtain the correct final remainder.

4.5 FLOATING-POINT REPRESENTATION

- A floating point number in computer registers consists of two parts: a mantissa m and an exponent e.
- The two parts represent a number obtained from multiplying m times a radix r raised to the value of e.

$$\mathbf{m \times r^e}$$

- The mantissa may be a fraction or an integer. The location of the radix point and the value of the radix r are assumed and are not included in the registers.
- For eg, the decimal number +6132.789 is represented in floating point with a fraction and exponent as follows:

Fraction	Exponent
+0.6132789	+04

- The above representation is equivalent to scientific notation $+0.6132789 \times 10^4$
- A floating point binary number is represented in a similar manner to floating point decimal number except that it uses base 2 for exponent.

- For eg, the binary number +1001.11 is represented as 8-bit fraction and 6-bit exponent as follows:

Fraction	Exponent
01001110	000100

- The above fraction has 0 in the left most position to denote positive.
- The above floating point number is equivalent to

  $m \text{ X } 2^e \rightarrow +(.1001110)_2 \text{ X } 2^{+4}$

- A floating-point number is said to be normalized if the most significant digit of the mantissa is nonzero.
- For example, the decimal number 350 is normalized but 00035 is not.
- Another example, the 8-bit binary number 00011010 is not normalized because of 3 leading 0's.
- A floating-point number that has a zero in the most significant bit position of the mantissa is said to have an underflow.
- To normalize the number that contains an underflow, it is necessary to shift the mantissa to the left and decrement the exponent until a non-zero appear in the first position.

4.5.1 Floating-point Arithmetic operations

- Arithmetic operations with floating-point numbers are more complicated than with fixed-point numbers and their execution takes longer 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.
- Consider the sum of the following floating-point numbers:

$$\begin{array}{r} \mathbf{.5372400 \text{ X } 10^{2}} \\ \mathbf{+\ .1580000 \text{ X } 10^{-1}} \end{array}$$

- It is necessary that the 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 the mantissas are stored 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.

$$\begin{array}{r} .5372400 \times 10^2 \\ +\ .0001580 \times 10^2 \\ \hline .5373980 \times 10^2 \end{array}$$

- 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:

$$\begin{array}{r} .56780 \times 10^5 \\ -\ .56430 \times 10^5 \\ \hline .00350 \times 10^5 \end{array}$$

- A floating-point number that has a 0 in the most significant position of the mantissa is said to have an underflow. To normalize a number that contains an underflow, it is necessary to shift the mantissa to the left and decrement the exponent until a nonzero digit appears in the first position.
- Floating-point multiplication and division do not require an alignment of the mantissas. The product can be formed by multiplying the two mantissas and adding the exponents.
- Division is accomplished by dividing the mantissas and subtracting the exponents.
- The exponent may be represented in any one of the three representations: signed-magnitude, signed-2's complement, or signed-1's complement. A fourth representation employed in many computers is known as a biased exponent.
- In this representation, the sign bit is removed from being 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.
- Each register is subdivided into two parts.
- The mantissa part has the same uppercase letter symbols as in fixed-point representation.
- The exponent part uses the corresponding lowercase letter symbol.

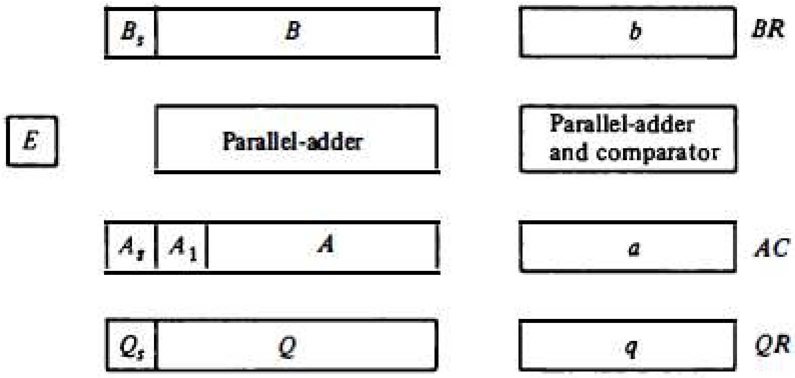


Figure 13: Registers for floating-point arithmetic operations

- A parallel-adder adds the two mantissas and transfers the sum into A and the carry into E.
- A separate parallel-adder is used for the exponents.
- 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 the binary point is assumed to reside to the left of the magnitude part.
- The numbers in the registers are assumed to be initially normalized. After each arithmetic operation, the result will be normalized.

4.5.2 Addition or subtraction of two floating-point numbers:

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**.**

- The normalization procedure ensures that the result is normalized prior to its transfer to memory. The flowchart for adding or subtracting two floating-point binary numbers is shown in Fig. below.
- If BR is equal to zero, the operation is terminated, with the value in the AC being the result. If AC is equal to zero, we transfer the content of BR into AC and also complement its sign if the numbers are to be subtracted.

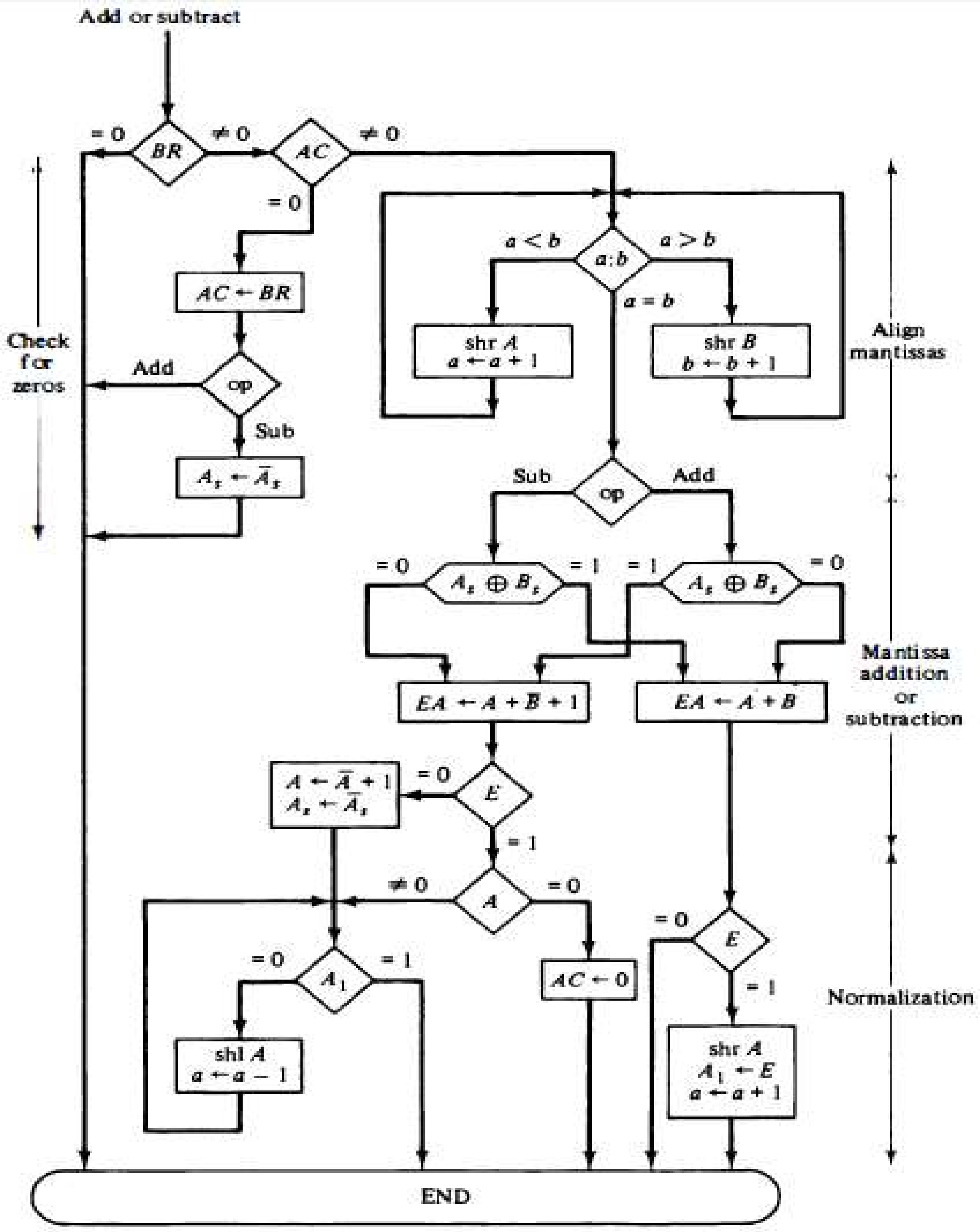


Figure 14: Addition and subtraction of floating-point numbers

- The magnitude comparator attached to exponents a and b provides three outputs that indicate their relative magnitude.
  i. If the two exponents are equal, we go to perform the arithmetic operation.
  ii. 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 the two exponents are equal.

- The magnitude part is added or subtracted depending 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 is equal to 1, the bit is transferred into A_1 and all other bits of A are shifted right.
- The exponent must be incremented to maintain the correct number. No underflow may occur in this case because the original mantissa that was not shifted during the alignment was already in a normalized position.
- If the magnitudes were subtracted, the result may be zero or may have an underflow. If the mantissa is zero, the entire floating-point number in the AC is made 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 A_1 is 0. In that case, the mantissa is shifted left and the exponent decremented. The bit in A_1 is checked again and the process is repeated until it is equal to 1. When $A_1 = 1$, the mantissa is normalized and the operation is completed.

4.5.3 Multiplication

- The multiplication of two floating-point numbers requires that we multiply the mantissas and add the exponents. No comparison of exponents or alignment of mantissas is necessary.
- The multiplication algorithm can be subdivided into four parts:
  1. Check for zeros.
  2. Add the exponents.
  3. Multiply the mantissas.
  4. Normalize the product.
- Steps 2 and 3 can be done simultaneously if separate adders are available for the mantissas and exponents.
- The two operands are checked to determine if they contain a zero. If either operand is equal to zero, the product in the AC is set to zero and the operation is terminated.

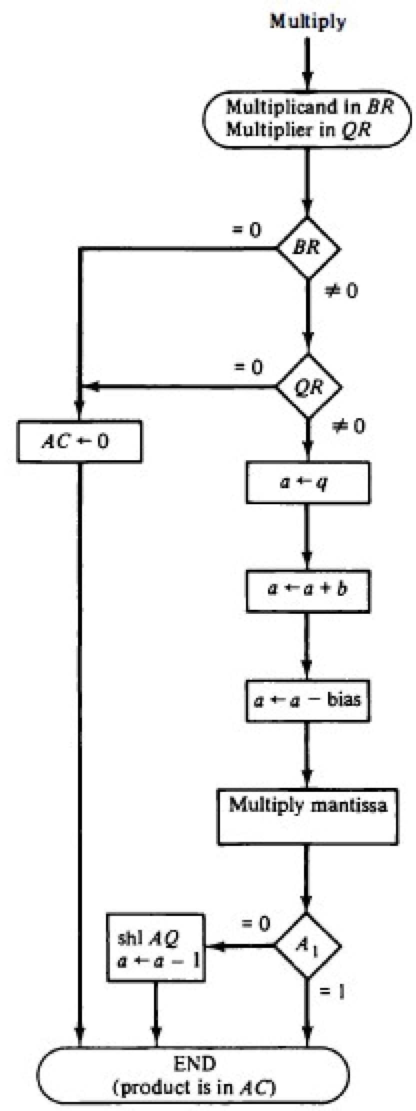


Figure 15: Multiplication of floating-point numbers

- If neither of the operands is equal to zero, the process continues with the exponent addition. The exponent of the multiplier is in q and the adder is between exponents a and b. It is necessary to transfer the exponents from q to a, add the two exponents, and transfer the sum into a.
- Since both exponents are biased by the addition of a constant, the exponent sum will have double this bias. The correct biased exponent for the product is obtained by subtracting the bias number from the sum.
- The multiplication of the mantissas is done as in the fixed-point case with the product residing in A and Q.

- The product may have an underflow, so the most significant bit in A is checked. If it is a 1, the product is already normalized. If it is a 0, the mantissa in AQ is shifted left and the exponent decremented.
- Note that only one normalization shift is necessary. The multiplier and multiplicand were originally normalized and contained fractions. The smallest normalized operand is 0.1, so the smallest possible product is 0.01. Therefore, only one leading zero may occur.

4.5.4 Division

- Floating-point division requires that the exponents be subtracted and the mantissas divided.
- The mantissa division is done as in fixed-point except that the dividend has a single-precision mantissa that is placed in the AC.
- The check for divide-overflow is the same as in fixed-point representation. However, with floating-point numbers the divide-overflow imposes no problems.
- If the dividend is greater than or equal to the divisor, the dividend fraction is shifted to the right and its exponent incremented by 1.
- For normalized operands this is a sufficient operation to ensure that no mantissa overflow will occur. The operation above is referred to as a dividend alignment.
- The division of two normalized floating-point numbers will always result in a normalized quotient provided that a dividend alignment is carried out before the division. Therefore, unlike the other operations, the quotient obtained after the division does not require normalization. The division algorithm can be subdivided into five parts:

  1. Check for zeros.
  2. Initialize registers and evaluates the sign.
  3. Align the dividend.
  4. Subtract the exponents.
  5. Divide the mantissas.
- The two operands are checked for zero. If the divisor is zero, it indicates an attempt to divide by zero, which is an illegal operation. The operation is terminated with an error message.

- An alternative procedure would be to set the quotient in QR to the most positive number possible (if the dividend is positive) or to the most negative possible (if the dividend is negative).If the dividend in AC is zero, the quotient in QR is made zero and the operation terminates.
- If the operands are not zero, we proceed to determine the sign of the quotient and store it in Q_s. The sign of the dividend in A_s is left unchanged to be the sign of the remainder.
- Next, the divisor exponent is subtracted from the dividend exponent. Since both exponents were originally biased, the subtraction operation gives the difference without the bias.

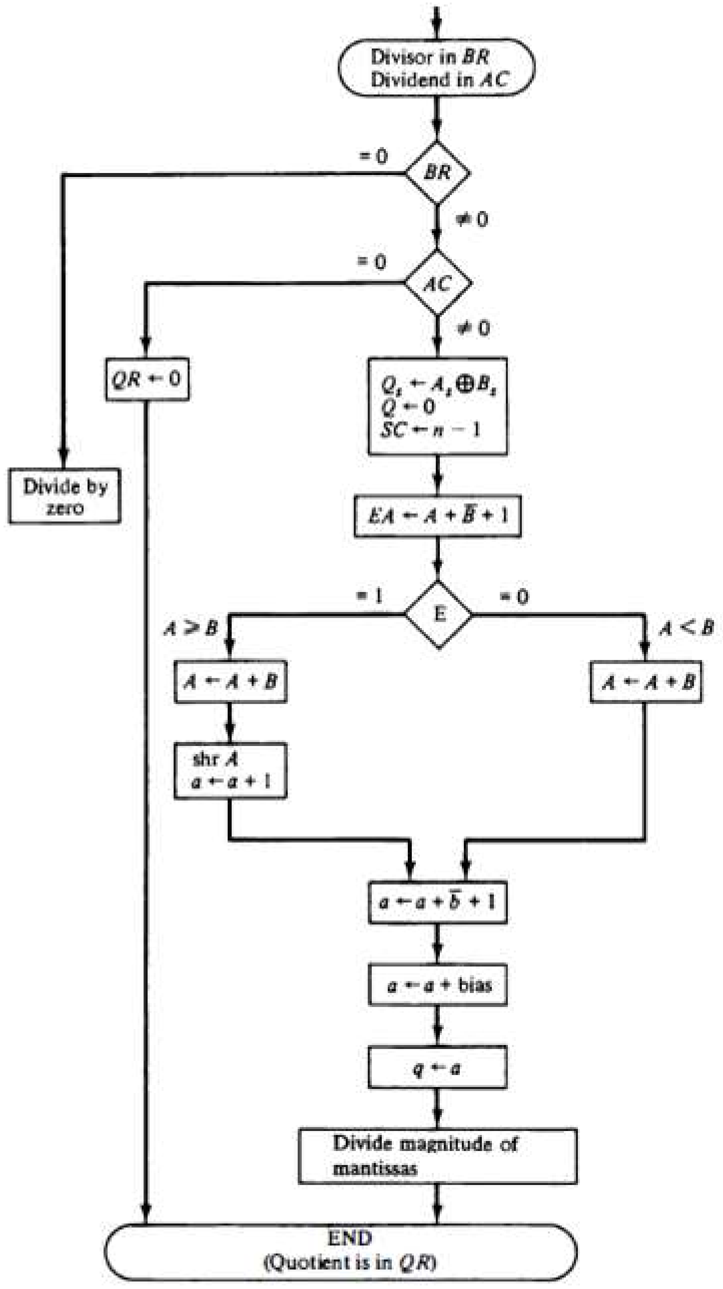


Figure 16: Division of floating point numbers

- The bias is then added and the result transferred into q because the quotient is formed in QR. The magnitudes of the mantissas are divided as in the fixed-point case. After the operation, the mantissa quotient resides in Q and the remainder in A.