UNIT-IV

Syllabus:

Computer Arithmetic: Addition and subtraction, Multiplication algorithms, division algorithms, floating-point arithmetic operations.

Introduction:

- Arithmetic instructions in digital computers manipulate data to produce results necessary for the solution of computational problems.
- These instructions perform arithmetic calculations and are responsible for the bulk of activity involved in processing data in a computer.
- The four basic arithmetic operations are **addition, subtraction, multiplication and division.** From these four bulk operations, it is possible to formulate other arithmetic functions and solve scientific problems by means of numerical analysis methods.
- An arithmetic processor is the part of a processor unit that executes arithmetic operations. The data type assumed to reside in **processor registers** during the execution of an arithmetic instruction is specified in the definition of the instruction. A:n arithmetic instruction may specify binary or decimal data, and in each case the data may be in fixed-point or floating-point form.
- We must be thoroughly familiar with the sequence of steps to be followed in order to carry out the operation and achieve a correct result. The solution to any problem that is stated by a finite number of well-defined procedural steps is called an **algorithm**.
- Usually, an algorithm will contain a number of procedural steps which are dependent on results of previous steps. A convenient method for presenting algorithms is a **flowchart**.

Addition and Subtraction:

- As we have discussed, there are three ways of representing negative fixed-point binary numbers: **signed-magnitude**, **signed-1's complement**, or **signed-2's complement**. Most computers use the signed-2's complement representation when performing arithmetic operations with integers.

i. Addition and Subtraction with Signed-Magnitude Data:

When the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed.

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

Algorithm: (Addition with Signed-Magnitude Data)

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

ii. When the signs of A and B are different, compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to be the same as A if A > B or the complement of the sign of A if A < B.

iii. If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

Algorithm: (Subtraction with Signed-Magnitude Data)

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

ii. When the signs of A and B are identical, compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to be the same as A if A > B or the complement of the sign of A if A < B.

iii. If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

Hardware Implementation:

To implement the two arithmetic operations with hardware, it is first necessary that the two numbers be stored in registers.

i. 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.

ii. The result of the operation may be transferred to a third register: however, a saving is achieved if the result is transferred into A and A_S. Thus A and A_S together form an accumulator register.

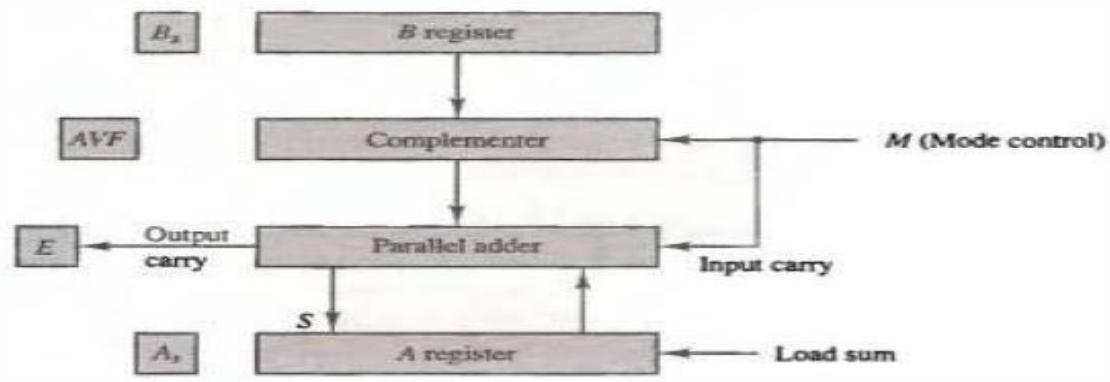
Consider now the hardware implementation of the algorithms above.

- First, **a parallel-adder** is needed to perform the microoperation A + B.
- Second, **a comparator circuit** is needed to establish if A > B, A = B, or A < B.
- Third, **two parallel-subtractor circuits** are needed to perform the microoperations A - B and B - A. The sign relationship can be determined from an exclusive-OR gate with A_S and B_S as inputs.

The below figure shows a block diagram of the hardware for implementing the addition and subtraction operations. It consists of registers A and B and sign flip-flops A_S and B_S.

- Subtraction is done by adding A to the 2' s complement of B. The output carry is transferred to flip-flop E, where it can be checked to determine the relative magnitudes of the two numbers.
- The add-overflow flip-flop AVF holds the overflow bit when A and B are added.

Figure (i): Hardware for addition and subtraction with Signed-Magnitude Data



The complementer provides an output of B or the complement of B depending on the state of the mode control M.

- When M = 0, the output of B is transferred to the adder, the input carry is 0, and the output of 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 $S = A + \bar{B} + 1$. This is equal to A plus the 2's complement of B, which isequivalent to the subtraction A - B.

Hardware Algorithm

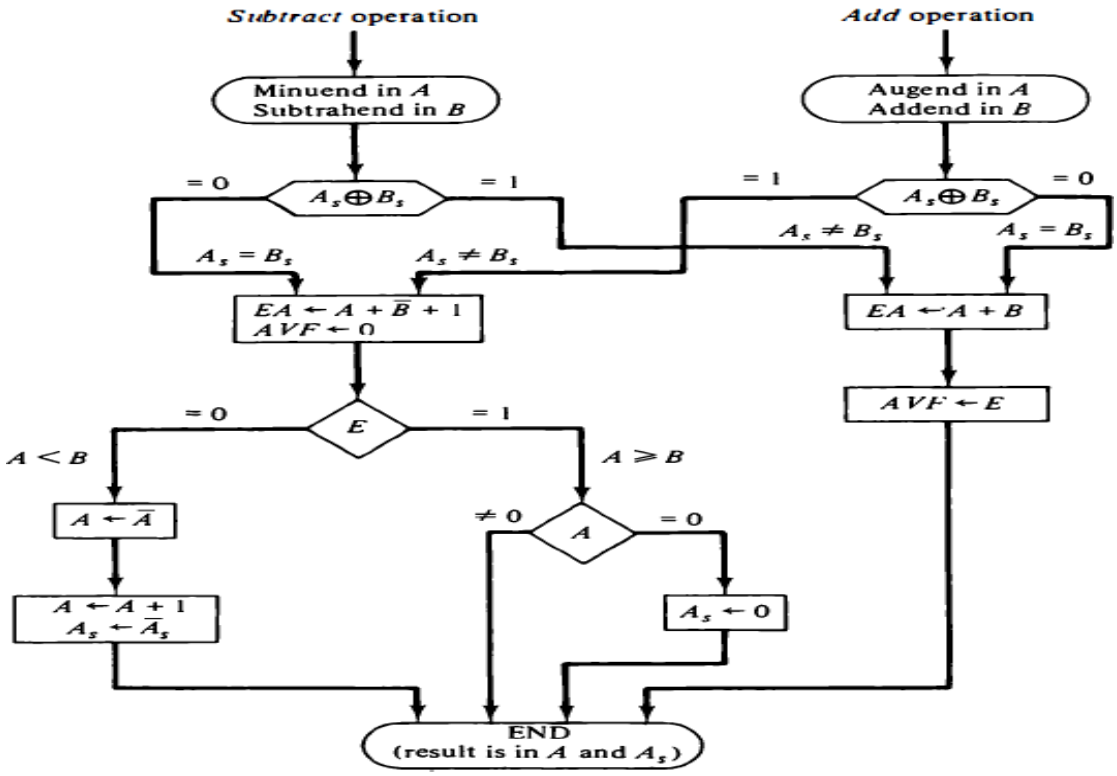
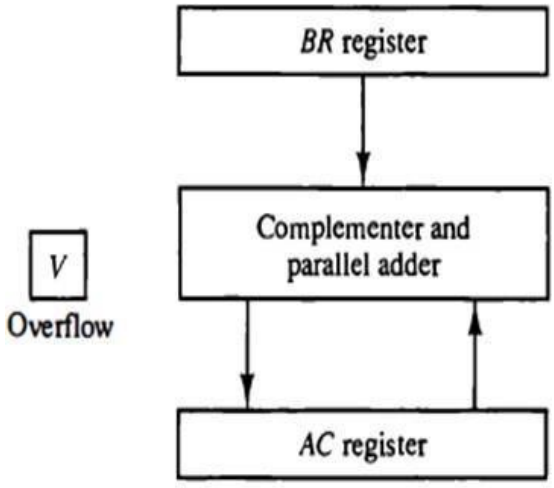


Figure (j): Flowchart for add and subtract operations

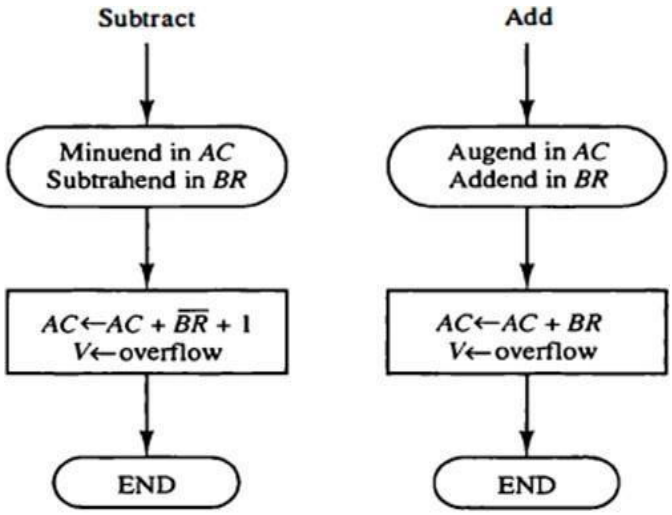
ii. Addition and Subtraction with Signed-2's Complement Data

- The register configuration for the hardware implementation is shown in the below Figure(a). We name the A register AC (accumulator) and the B register BR. The leftmost bit in AC and BR represent 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 to 1 if there is an overflow. The output carry in this case is discarded.
- The algorithm for adding and subtracting two binary numbers in signed-2' s complement representation is shown in the flowchart of Figure(b). 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.
- Comparing this algorithm with its signed-magnitude counterpart, we note that it is much simpler to add and subtract numbers if negative numbers are maintained in signed-2' s complement representation.

Example:+32 is represented as 00100001 and -32 as 11011111. Note that 11011111 is the 2's complement of 00100001.



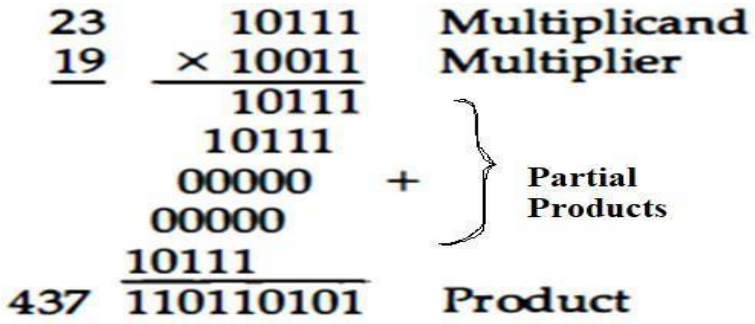
Figure(a): Hardware for addition & subtraction of 2's complement numbers



Figure(b): Algorithm for adding & subtracting of 2's complement numbers

Multiplication Algorithms:

Multiplication of two fixed-point binary numbers in signed-magnitude representation is done with paper and pencil by a process of successive **shift** and **adds** operations. This process is best illustrated with a numerical example.



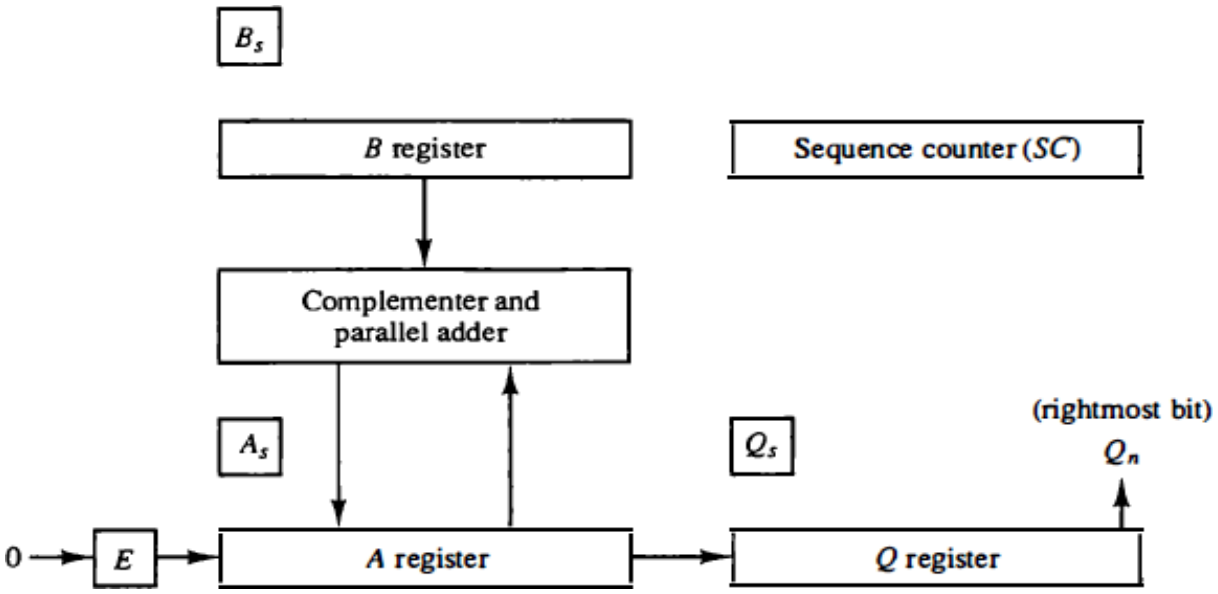
The process of multiplication:

- It consists of looking at successive bits of the multiplier, least significant bit first.
- If the multiplier bit is a 1, the multiplicand is copied down; otherwise, zeros are copied down.
- The numbers copied down in successive lines are shifted one position to the left from the previous number.
- Finally, the numbers are added and their sum forms the product.

The sign of the product is determined from the signs of the multiplicand and multiplier. If they are alike, the sign of the product is **positive**. If they are unlike, the sign of the product is **negative**.

Hardware Implementation for Signed-Magnitude Data

- The registers A, B and other equipment are shown in Figure (a). The multiplier is stored in the Q register and its sign in Qs. The sequence counter SC is initially set to a number equal to the number of bits in the multiplier. 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.



Figure(k): Hardware for multiply operation.

- Initially, the **multiplicand** is in register B and the **multiplier** in Q, Their corresponding signs are in Bs and Qs, respectively
- 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.
- The least significant bit of A is shifted into the most significant position of Q, the bit fromE 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.

Hardware Algorithm:

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

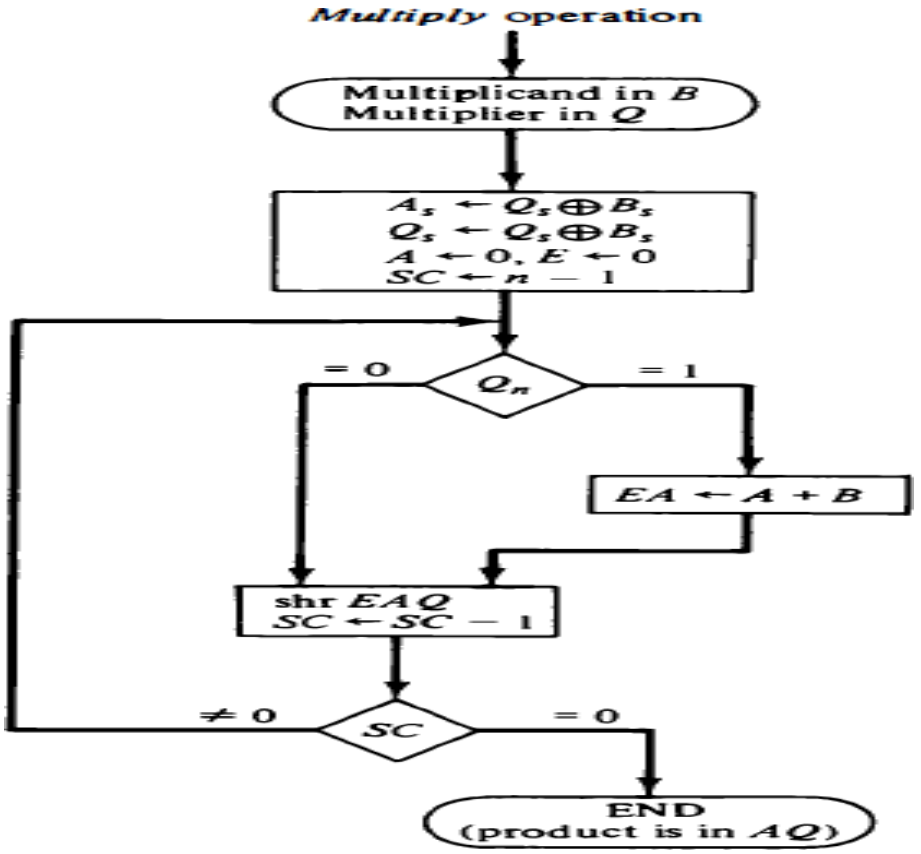
After the initialization, the low-order bit of the multiplier in Qn is tested.

i. If it is 1, the multiplicand in B is added to the present partial product in A .

ii. If it is 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.

The final product is available in both A and Q, with A holding the most significant bits and Q holding the least significant bits. A flowchart of the hardware multiply algorithm is shown in the below figure (l).



Figure(l): Flowchart for multiply operation

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				

Figure (m): Numerical Example of multiplication

Booth Multiplication Algorithm:(multiplication of 2's complement data):

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

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

Hardware implementation of Booth algorithm Multiplication:

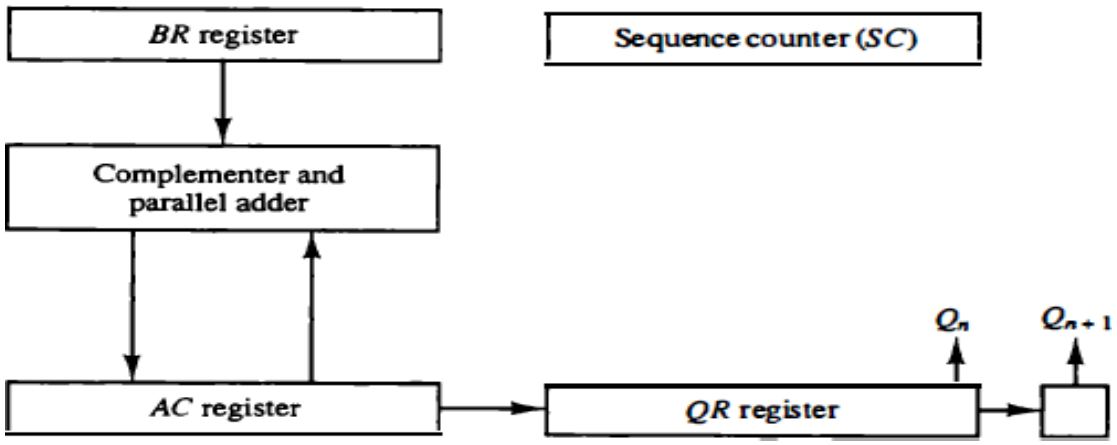


Figure (n): Hardware for Booth Algorithm

The hardware implementation of Booth algorithm requires the register configuration shown in figure (n). This is similar addition and subtraction hardware except that the sign bits are not separated from the rest of the registers. To show this difference, we rename registers A, B, and Q, as AC, BR, and QR, respectively. Q_n designates the least significant bit of the multiplier in register QR. 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 (o).

Hardware Algorithm for Booth Multiplication:

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.

i. 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.

ii. 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.

iii. When the two bits are equal, the partial product does not change.

iv. 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.

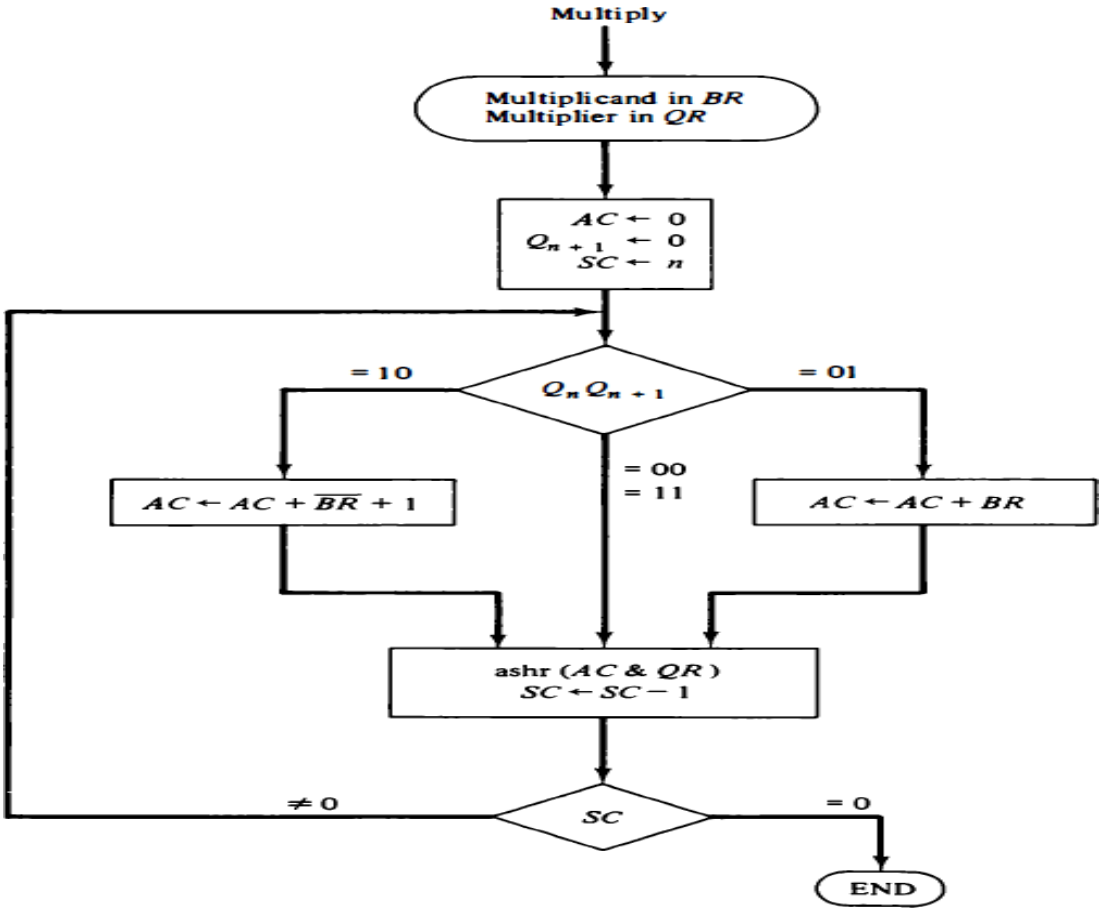


Figure (o): Booth Algorithm for multiplication of 2's complement numbers

Example: multiplication of (- 9) x (- 13) = + 117 is shown below. Note that 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

Figure (p): Example of Multiplication with Booth Algorithm.

Division Algorithms:

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

The division process is illustrated by a numerical example in the below figure (q).

- The divisor B consists of five bits and the dividend A consists of ten bits. The five most significant bits of the dividend are **compared** with the divisor. Since the 5-bit number is smaller than B, we try again by taking the sixth 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. The divisor is then shifted once to the right and subtracted from the dividend.
- The difference is called a **partial remainder** because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder. The process is continued by comparing a partial remainder with the divisor.
- If the partial remainder is greater than or equal to the divisor, the quotient bit is equal to 1. The divisor is then shifted right and subtracted from the partial remainder.

- If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction is needed. The divisor is shifted once to the right in any case. Note that the result gives both a quotient and a remainder.

```
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 (q): Example of Binary Division

Hardware Implementation for Signed-Magnitude Data:

The hardware for implementing the division operation is identical to that required for multiplication.

- ✓ The divisor is stored in the B register and the double-length dividend is stored in registers A and Q. The dividend is shifted to the left and the divisor is subtracted by adding its 2's complement value. The information about the relative magnitude is available in E.
- ✓ If E = 1, it signifies that A≥B. A quotient bit 1 is inserted into Q, and the partial remainder is shifted to the left to repeat the process.
- ✓ If E = 0, it signifies that A < B so the quotient in Qn remains a 0. The value of B is then added to restore the partial remainder in A to its previous value. The partial remainder is shifted to the left and the process is repeated again until all five quotient bits are formed.
- ✓ Note that while the partial remainder is shifted left, the quotient bits are shifted also and after five shifts, the quotient is in Q and the final remainder is in A.

The sign of the quotient is determined from the signs of the dividend and the divisor. If the two signs are alike, the sign o f the quotient is **plus**. If they are unalike, the sign is **minus**. The sign of the remainder is the same as the sign of the dividend.

Divide Overflow

- The division operation may result in a quotient with an overflow. This is not a problem when working with paper and pencil but is critical when the operation is implemented with hardware. This is because the length of registers is finite and will not hold a number that exceeds the standard length.
- To see this, consider a system that has 5-bit registers. We use one register to hold the divisor and two registers to hold the dividend. From the example shown in the above, we note that the quotient will consist of six bits if the five most significant bits of the dividend constitute a number greater than the divisor. The quotient is to be stored in a standard 5-bit register, so the overflow bit will require one more flip-flop for storing the sixth bit.
- This divide-overflow condition must be avoided in normal computer operations because the entire quotient will be too long for transfer into a memory unit that has words of standard length, that is, the same as the length of registers.
- This condition detection must be included in either the hardware or the software of the computer, or in a combination of the two.

When the dividend is twice as long as the divisor,

i. A divide-overflow condition occurs if the high-order half bits of the dividend constitute a number greater than or equal to the divisor.

ii. 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**.

Hardware Algorithm:

1. The dividend is in A and Q and the divisor in B . The sign of the result is transferred into Qs to be part of the quotient. A constant is set into the sequence counter SC to specify the number of bits in the quotient.

2. 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.

3. 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 Qn for the quotient bit.

4. 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 ≥ B; therefore, Qn is set to 1 . If E = 0, it signifies that A < B and the original number is restored by adding B to A . In the latter case we leave a 0 in Qn.

This process is repeated again with registers EAQ. After n times, the quotient is formed in register Q and the remainder is found in register A

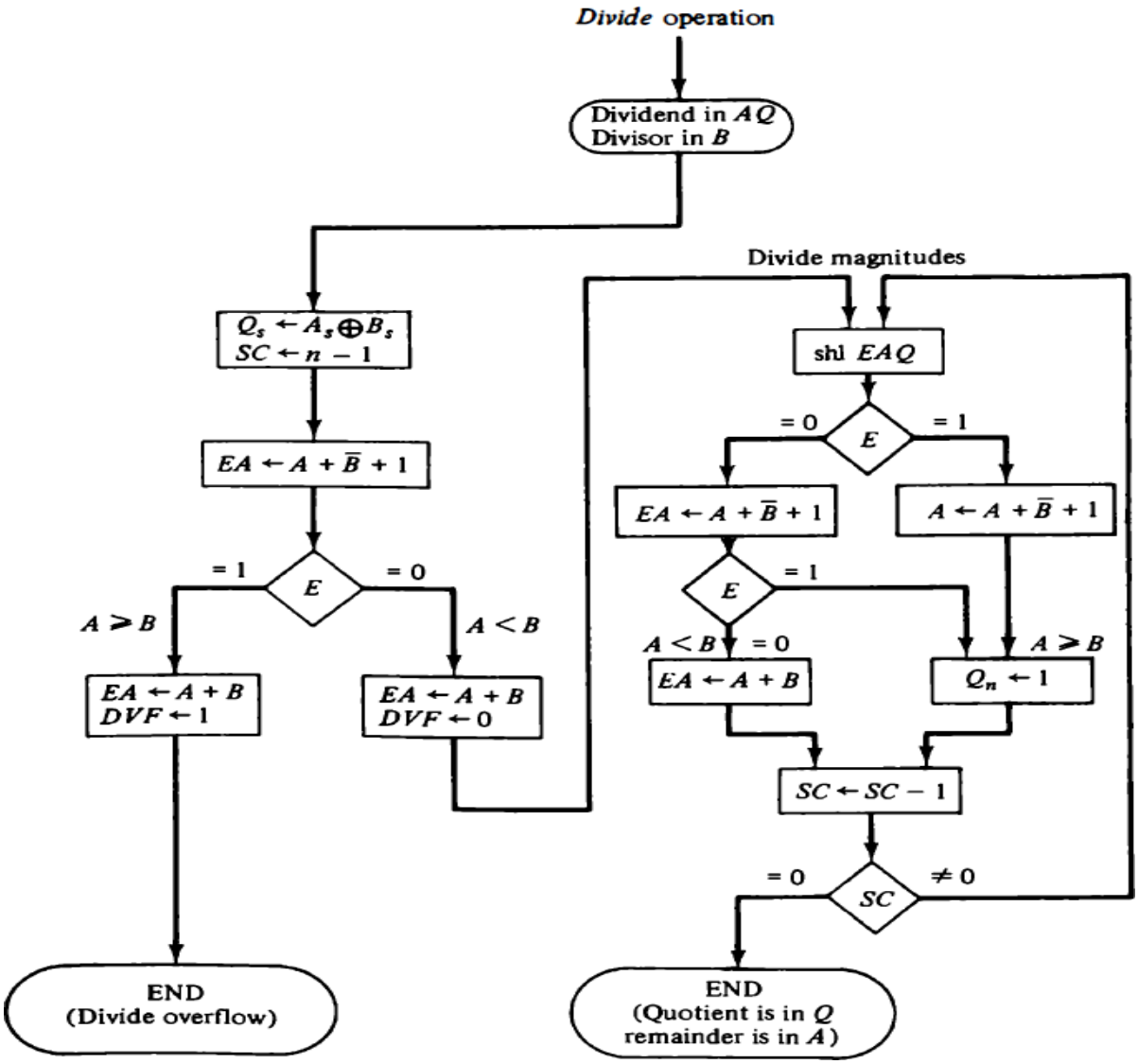


Figure (r): Flowchart for Divide operation

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

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

Figure (s): Example of Binary Division

Floating-point arithmetic operations

The most common way is to specify them by a real declaration statement as opposed to fixed–point numbers, which are specified by an integer declaration statement. Any computer that has a compiler for such high–level programming language must have a provision for handling floating–point arithmetic operations. The compiler must be designed with a package of floating–point software subroutines. The hardware method is more expensive, it is so much more efficient than the software method.

Basic Considerations

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 time a radix r raised to the value of e; thus

$$m \times r^e$$

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. The decimal number 537.25 is represented in a register with m = 53725 and e = 3 and is interpreted to represent the floating –point number

$$.53725 \times 10^3$$

A floating – point number is normalized if the most significant digit of the mantissa is nonzero. In this way the mantissa contains the maximum possible number of significant digits. A zero cannot be normalized because it does not have a nonzero digits. A zero cannot be normalized 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 that can be accommodated in a given register.

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. The alignment is done by shifting one mantissa while its exponent is adjusted until it is equal to the other exponent. Consider the sum of the following floating – point numbers:

$$.5372400 \times 10^2$$
$$+ .1580000 \times 10^{-1}$$

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. 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. After this is done, the mantissas can be added :

$$.5372400 \times 10^2$$
$$+.0001580 \times 10^2$$
$$.5373980 \times 10^2$$

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:

$$.56780 \times 10^5$$
$$-.56430 \times 10^5$$
$$.00350 \times 10^5$$

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 operations performed with the exponents are compare and increment (for aligning the mantissas), add and subtract (for multiplication and division), and decrement (to normalize the result). The exponent may be represent in any one of the three representation : 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.

The advantage of biased exponents is that they contain only positive numbers. It is then simpler to compare their relative magnitude without being concerned with their signs. As a consequence, a magnitude comparator can be used to compare their relative magnitude during the alignment of the mantissa and the smallest possible exponent.

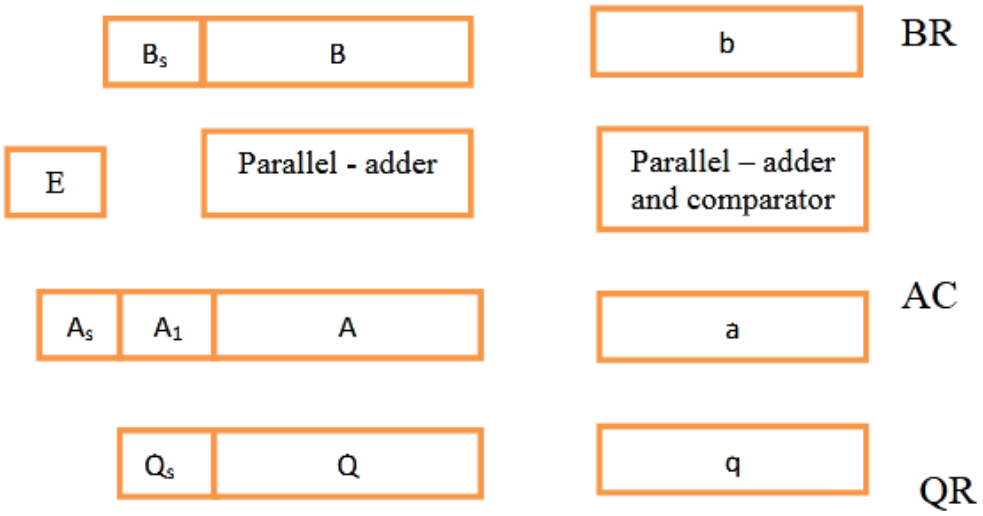
Register Configuration

The register configuration for floating – point operations is quite similar to the layout for fixed – point operations.

There are three registers, 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.

Each floating – point number has a mantissa in signed magnitude representation and a biased exponent. Thus the AC has a mantissa

Figure 5.13 Registers for floating–point arithmetic operation



whose sign is in A_s and a magnitude that is in A. The exponent is in the part of the register denoted by the lowercase letter symbol a. The diagram shows explicitly the most significant bit of A, labeled by A_1. The bit in this position must be a 1 for the number to be normalized. Note that the symbol AC represents the entire register, that is, the concatenation of A_s, A and a. Register BR is subdivided into Bs, B and b, and QR into Qs, Q, and q. 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. Since the exponents are biased, they do not have a distinct sign bit but are represented as a biased positive quantity. The floating–point numbers are so large that the chance of an exponent overflow is very remote, the exponent overflow will be neglected. The exponents are also connected to a magnitude comparator that provides three binary outputs to indicate their relative magnitude.

The number in the mantissa will be taken as a fraction, so 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. Thus all floating – point operands coming from and going to the memory unit are always normalized.

Addition and Subtraction

During addition or subtraction, the two floating–point operands are 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

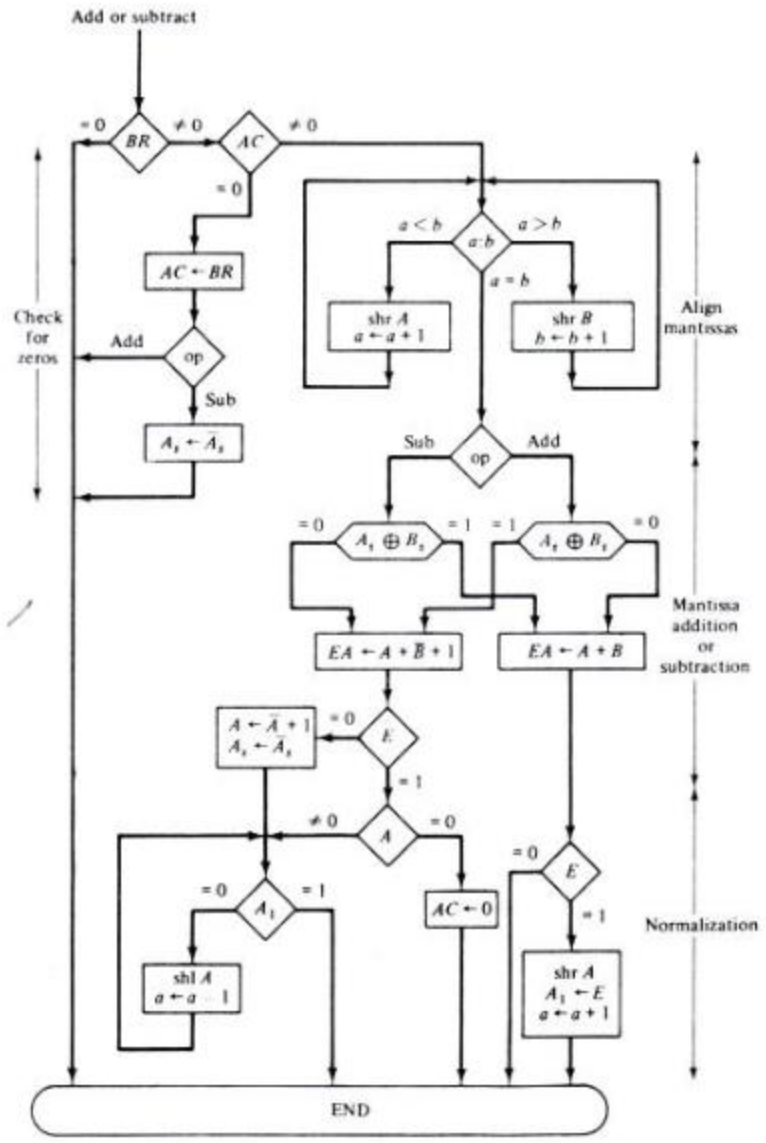
A floating–point number that is zero cannot be normalized. If this number is used during the computation, the result may also be zero. Instead of checking for zeros during the normalization process we check for zeros at the beginning and terminate the process if necessary. The alignment of the mantissas must be carried out prior to their operation. After the mantissas are added or subtracted, the result may be normalized. 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.5.14. 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. If neither number is equal to zero, we proceed to align the mantissas.

The magnitude comparator attached to exponents a and b provides three outputs that indicate their relative magnitude. 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 the two exponents are equal.

The addition and subtraction of the two mantissas is identical to the fixed - point addition and subtraction of the two mantissas is identical to the fixed - point addition and subtraction algorithm. 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 the 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.



. Figure 5.14 Addition and subtraction of floating point numbers

Multiplication

The multiplication of the mantissas is performed in the same way as in fixed–point to provide a double – precision product. The double–precision answer is used in fixed–point numbers to increase the accuracy of the product. In floating–point, the range of a single precision mantissa combined with the exponent is usually accurate enough so that only single precision numbers are maintained. Thus the half most significant bits of the mantissa product and the exponent will be taken together to form a single precision floating–point product.

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. 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. Overflow cannot occur during multiplication, so there is not need to check for it.

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.

Although the low – order half of the mantissa is in Q, we do not use it for the floating – point product. Only the value in the AC is taken as the product.

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. For integer representation, a single – precision dividend must be placed in register Q and register A must be cleared. The zeros in A are to the left of the binary point and have no significance. In fraction representation, a single – precision dividend is placed in register A and register Q is cleared. The zeros in Q are to the right of the binary point and have no significance.

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.

The division algorithm can be subdivided into five parts

1. Check for zeros
2. Initialize registers and evaluate 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. The Q register is cleared and the sequence counter SC is set to a number equal to the number of bits in the quotient.

The dividend alignment is similar to the divide – overflow check in the fixed – point operation. The proper alignment requires that the fraction dividend be smaller than the divisor. The two fractions are compared by a subtraction test. The carry in E determines their relative magnitude. The dividend fraction is restored to its original value by adding the divisor. If $A \geq B$, it is necessary to shift A once to the right and increment the dividend exponent. Since both operands are normalized, this alignment ensures that $A < B$.