COMPUTER ORGANIZATION AND ARCHITECHTURE

UNIT-3

Department of Computer Science and Engineering

Floating point number representation:

Fixed point numbers represents integers and floating point numbers represent real numbers.

Numeric Format:

A number expressed in scientific notation has **a sign, a fraction or significant**(or mantissa) and an **exponent**.

Ex: The number is : -1234.5678

Scientific notation is : $-1.2345678 * 10^3$. Here the sign is negative, the significant is 1.2345678 and exponent is 3 and the base is 10. Computers use base as 2.

Disadvantages of Scientific notation:

Most of the numbers can be expressed in many different ways. Ex: -1.2345678*10^3 = -1234567.8* 10^{-3} = etc. Computers are more efficient and have much simpler hardware if each number is uniquely represented.

Normalization as Solution to the problem:

The floating point number must be normalzed, that is, each number's significant is a fraction with no leading zeros. Thus the only valid floating point representation for - 1234.5678 is -.12345678* 10^4 . Note **IEEE 754** uses an exception for this rule.

Special cases:

The number zero has only zeros in its significand and can not be normaliuzed.For this reason a special value is assigned to zero. Arithmetic algorithms must explicitly check for zero values and treat them as special cases. +∞ and - ∞ also have special representation and require special treatment.

NaN:

NaN means Not a Number. It represents the result of illegal operations, such as ∞ ÷ ∞ or taking the square root of a negative number. As with zero and infinity, NaN requires a special treatment in floating point arithmetic algorithms.

A predefined format for computer storage of floating point number:

Each number is stored in it's normal form.

Take the number: X = - 1234.5678 ; That is X= X_S $X_F X_E$

X_S is the sign of X; X_F is it's significand and X_E is it's exponent

Since the radix point is located to the left of the most significant bit of the significand , the radix point is not stored.Thus the value X = -1234.5678 would be stored as $X_{S\,=}$ 1, X_F = 12345678 and X_E = 4

Biasing:

In the above representation foe exponent , there is no sign bit for exponent. We can use 2's complement form but prevalent practice is to use **biasing.**

If X_E has 4 bits, then it can represent 16 items. That is the numbers from -8 to +7. To do this, a set bias value is added to the actual exponent. The result is ten stored in $X_{E..}$For this the bias should be set to 8.

The smallest possible exponent, -8, is represented as -8+bias = -8+8=0 or 0000 in binary.

The largest possible exponent, +7, is represented as +7+ bias =+7+8= 15= 1111 in binary. The arithmetic algorithms must account for the bias when generating their results.

Characteristics of floating point numbers:

The characteristics are 1. Precision 2. Gap 3. Range

Precision:

It characterizes how precise a floating point value can be. I**t is defined as the number of bits in the significand**. The greater the number of bits in the significand, the greater is the CPU's precision and the more precise is it's value. Many CPUs have 2 representations for floating point numbers. They are called single precision and double precision here double precision has twice the number of bits.

Gap:

The gap is the difference between two adjacent values. It's value depends on the value of the exponent.

Take the number: X = .10111010 * 2^3 .

It's adjacent values are : .10111001 * 2^3 and .10111011* 2^3 .

Each number produce a gap of .00000001* 2^3 .

In general the gap for floating point value X can be expressed as $\mathbf{2}^{(\text{ Xe-precision})}$

Range:

The range of a floating point representation is bounded by it's smallest and largest possible values.

Overflow and underflow;

Overflow occurs when an operation produces a result that can not be stored in computers's floating point registers. Underflow occurs when an operation produces a result between zero and either the positive or negative smallest possible value.

IEEE 754 Floating point standard:

This standard specifies 2 precision for floating point numbers which are called single precision and double precision floating point representations.

Single Precision Format:

This format has 32 bits. 1 bit for sign; 8 bits for the exponent; 23 for the significand. The significand also includes an implied 1 to the left of its radix point(except for special values and denormalized numbers).

Error Detection codes:

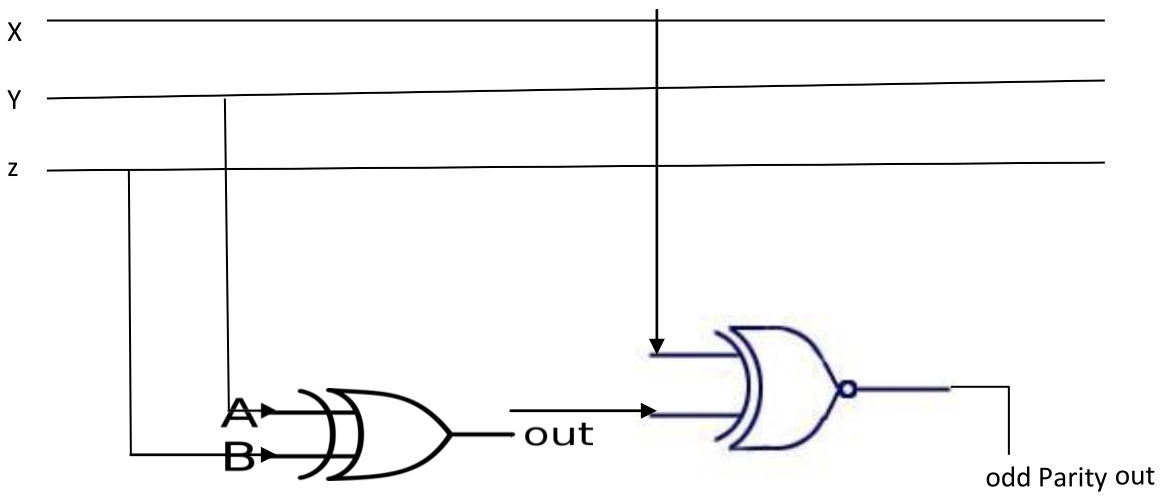
Information is stored as binary codes and are transmitted by serial or parallel communication. During transmission noise is added to the signal and it may change binary bits in the code from 1 to 0, and vice versa. An error detection code is a **binary code that detects digital errors during transmission.** The detected errors can not be corrected but their presence is indicated.

Parity bit:

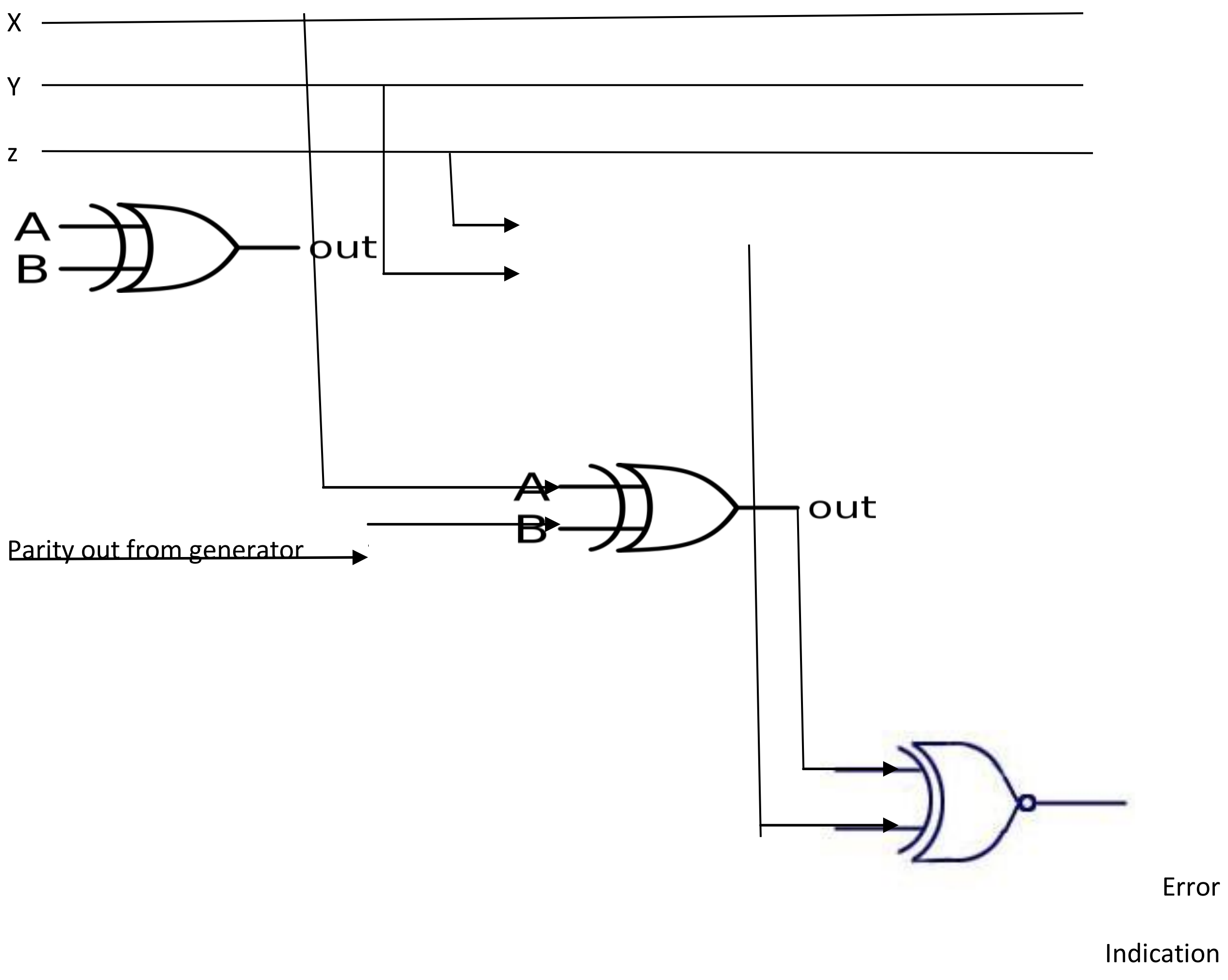
The most common error detection code used is the parity bit. A parity bit is an extra bit included with a binary message to make the total number of 1's either odd or even. If the message consists of n bits , then the error detection code consists of n+1 bits. If the bit added to the message makes the sum of 1's odd in the error detection code, then the scheme is called odd-parity. If the sum of bits is even , the scheme is called even parity scheme.

Message xyz	P(odd)	P(even)	Error detection code, odd parity	Error detection code, even parity
000	1	0	0001	0000
001	0	1	0010	0011
010	0	1	0100	0101
011	1	0	0111	0110
100	0	1	1000	1001
101	1	0	1011	1011
110	1	0	1101	1101
111	0	1	1110	1111

Parity Generator and Parity Checker:



Parity Checker:



The circuit arrangement checks the occurrence of error any odd number of times. An even number of errors is not detected.

We note that P(even) function is the exclusive –OR x,y,z because it is equal to 1 when either one or all 3 of the variables are equal to 1. The P(odd) function is the complement of the P(even) function.

Assume at the sending end the message bits and odd parity bit is generated. The EX-OR gates generate P(even) function and to generate P(odd), the complement of P(even) is used.

The 4 bits transmitted has an odd number of I's. If an error occurs during transmission, then the number of 1's become even. Hence parity checker checks for even parity.

COMPUTER ARITHMETIC:

Addition, subtraction, multiplication 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.

Arithmetic Processor:

It is the part of a processor unit that executes arithmetic operations. The arithmetic instructions definitions specify the data type that should be present in the registers used . The arithmetic instruction may specify binary or decimal data and in each case the data may be in fixed-point or floating point form.

Fixed point numbers may represent integers or fractions. The negative numbers may be in signed-magnitude or signed- complement representation. The arithmetic processor is very simple if only a binary fixed point add instruction is included. It would be more complicated if it includes all four arithmetic operations for binary and decimal data in fixed and floating point representations.

Algorithm:

Algorithm can be defined as a finite number of well defined procedural steps to solve a problem. Usually, an algorithm will contain a number of procedural steps which are dependent on results of previous steps. A convenient method for presenting an algorithm is a flowchart which consists of rectangular and diamond –shaped boxes. The computational steps are specified in the rectangular boxes and the decision steps are indicated inside diamond-shaped boxes from which 2 or more alternate path emerge.

Addition and Subtraction:

3 ways of representing negative fixed point binary numbers:

1. Signed-magnitude representation---- used for the representation of mantissa for floating point operations by most computers.
2. Signed-1's complement
3. Signed -2's complement—Most computers use this form for performing arithmetic operation with integers

Addition and subtraction algorithm for signed-magnitude data

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 te 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 te sign of the result positive.

Hardware Implementation:

Let A and B are two registers that hold the numbers.

A_S and B_S are 2, flip-flops that hold sign of corresponding numbers. The result is stored In A and A_S .and thus they form Accumulator register.

We need to perform micro operation, A+ B and hence a parallel adder.

A comparator is needed to establish if A> B, A=B, or A<B.

We need to perform micro operations A-B and B-A and hence two parallel subtractor.

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:

1. We know subtraction can be done by complement and add.
2. 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.

Hardware forsigned-magnitude addition and subtraction:

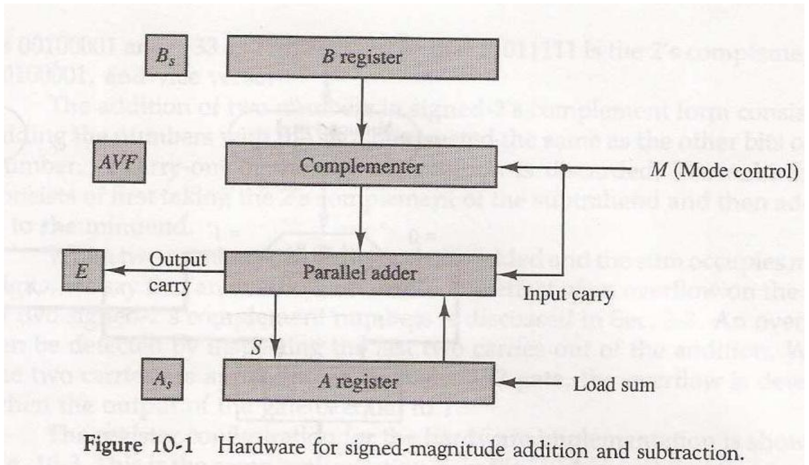


Figure 10-1 Hardware for signed-magnitude addition and subtraction.

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 B. It consists of EX-OR gates and the parallel adder consists of full adder circuits. The M 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:

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.

[the subtraction of 2 n-digit un signed numbers M-N (N≠0) in base r can be done as follows:

1. Add minuend M to thee r's complement of the subtrahend N. This performs M-N +r^n.
2. If M ≥ N, The sum will produce an end carry r^nwhich is discarded, and what is left is the result M-N.
3. If M< N, the sum does not produce an end carry and is equal to r^n–(N-M), which is the r's complement of the sum and place a negative sign in front.]

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 A_S 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 A_S isrequired.

However When A < B, the sign of the result is the complement of original sign of A. Hence The complement of A_S stored in A_S.

Final Result: A_S A

Flow chart for ADD and Subtract operations:

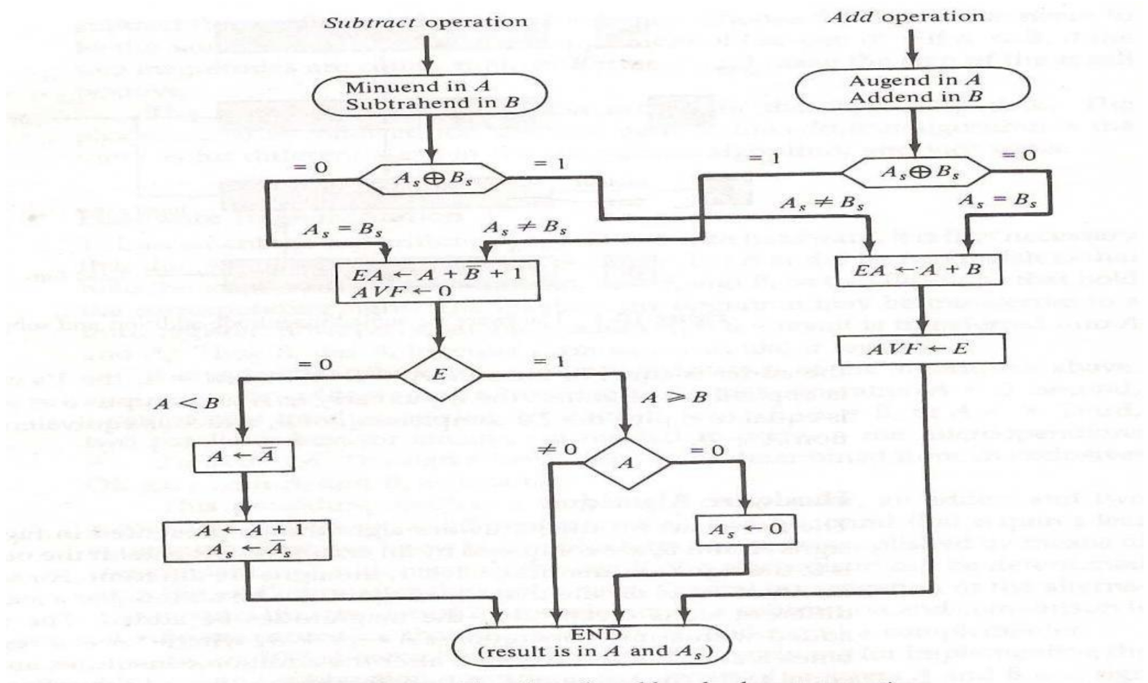


Figure 10-2 Flowchart for add and subtract operations.

Addition and Subtraction with signed-2's complement Data.:

Arithmetic Addition:

This method does not need a comparison or subtraction but only addition and complementation. The procedure is as below:

1. **Represent the negative numbers in 2's complement form.**
2. **Add the two numbers including the sign bits and discard any carry out of sign bit position.**
3. **The overflow bit V is set to 1 if there is a carry into sign bit and no carry out of sign bit or if there is a no carry into sign bit and a carry out of sign bit. Otherwise it is set to zero.**
4. **If the result is negative, take the 2's complement of the result to get a correct negative result.**

Arithmetic Subtraction:

1. **Represent the negative numbers in 2's complement form.**
2. **Take the 2's complement of the subtrahend including the sign bit and add it to the minuend including the sign bit.**
3. **The overflow bit V is set to 1 if there is a carry into sign bit and no carry out of sign bit or if there is a no carry into sign bit and a carry out of sign bit. Otherwise it is set to zero.**
4. **Discard the carry out of the sign bit position.**

Note: A subtraction operation can be changed to an addition operation if the sign of the subtrahend is changed.

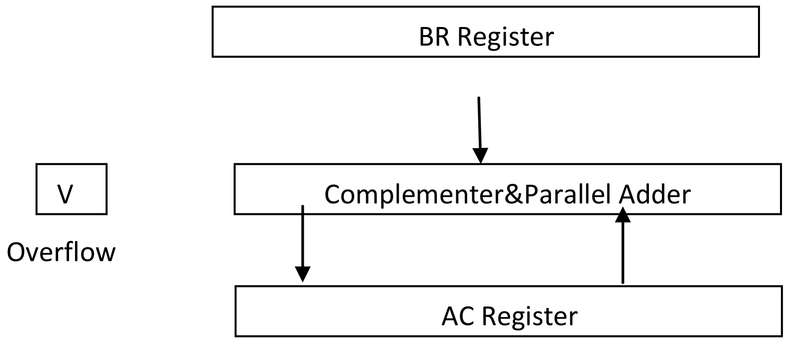


Fig: Hardware for Signed 2/s complement for addition/ subtractioin.

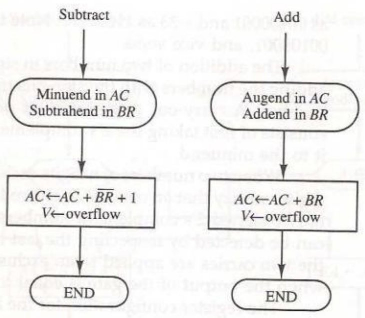


Figure 10-4 Algorithm for adding and subtracting numbers in signed-2's complement representation.

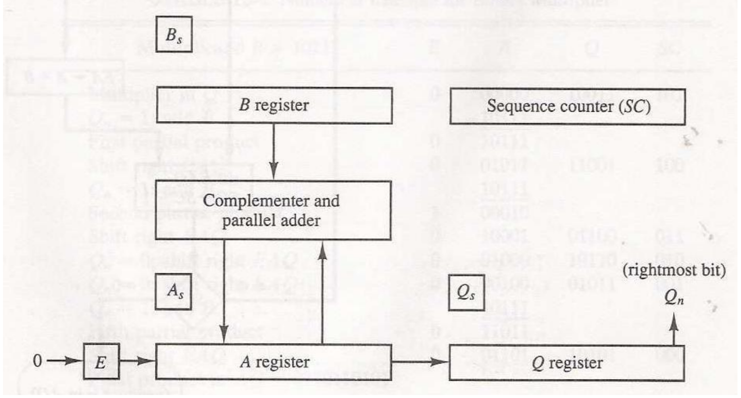
Multiplication Algorithm:

Hardware implementation of multiplication of numbers in signed – magnitude form:

1. A adder is provided to add two binary numbers and the partial product is accumulated in a register.
2. Instead of shifting the multiplicand to the left, the partial product is shifted to the right, which result in leaving the partial product and the multiplicand in the required relative positions.
3. When the corresponding bit of the multiplier is zero, there is no need to add all zeros to the partial product, since it will not alter it's value.

The hardware consists of 4 flip flops, 3 registers, one sequence counter , an adder and complementer.

Figure 10-5 Hardware for multiply operation.



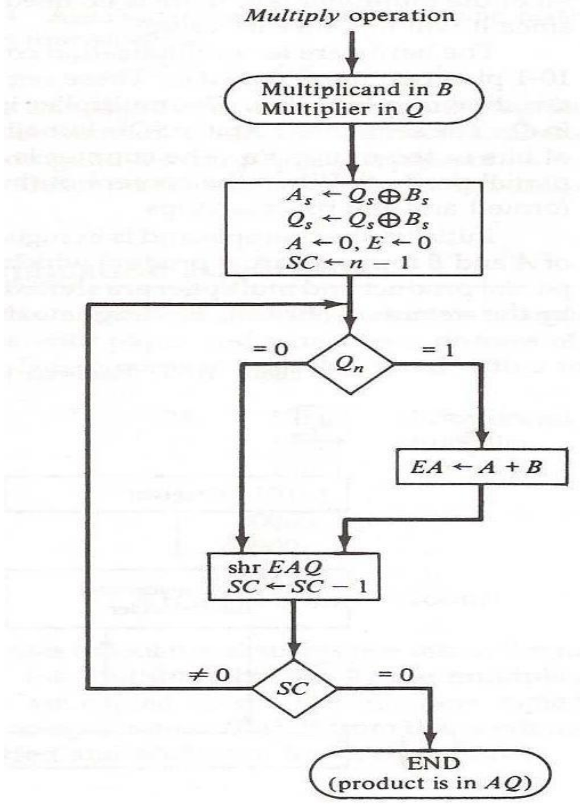
Q register&Q_S flip flop	:	contains multiplier & Its sign
Sequence counter	:	It is set to a value equal to the number of bits in the multiplier
B Register& B_S flipflop	:	It contains the multiplicand,& its sign
A Register, E Flip flop	:	Initialized to ' 0'. A_S denotes sign of partial product
EA Register	:	hold partial product, with carry generated in addition being shifted to E .
Qn	:	Rightmost bit of the multiplier; AQ : will contain the final product.

As **AQ** represent product register, both A_S $Q_S$represent the sign of the partial product or product.
The number to be multiplied are stores in memory as n bit sign magnitude numbers and when transferred to register msb bit go to sign flipflop and remaining n-1 bits go to registers. Hence SC is initially set to n-1.
Let the lower order bit of the multiplier in $Q_n$tested.
If it is 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 it's 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.

Flowchart for multiply operation:

Figure 10-6 Flowchart for multiply operation.



Numerical Example for the above algorithm:

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 Fifth Partial Product Shift Right EAQ	 0 0	10111 11011 01101	 10101	 000
Final Product in AQ AQ = 0110110101				

Booth Multiplication Algorithm:

Multiplication of signed- 2's complement integers:

This algorithm uses the following facts.

1. A string of 0's in the multiplier requires no addition but just shifting.
2. A string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as 2^{k+1} - $2^{m.}$

Example: Consider the binary number: 001110(+14)

The number has a string of 1's from 2^3 to 2^1 . Hence k = 3 and m= 1. As other bits are 0's, the number can be represented as 2^{k+1} - 2^m = 2^4 – 2^1 = 16-2 = 14. Therefore the multiplication M * 14 , where M is the multiplicand and 14 the multiplier can be done as Mx 2^4 –M x $2^{1.}$

This can be achieved by shifting binary multiplicand M four times to the left and subtracting M shifted left once which is equal to (Mx 2^4 –M x $2^{1.}$).

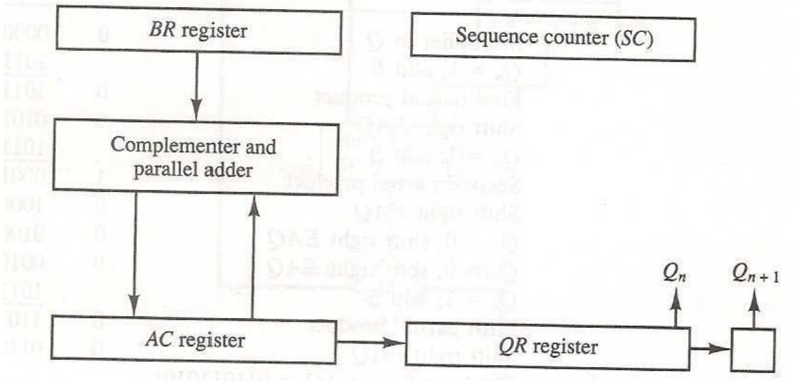
Shifting and addition/subtraction rules for multiplicand in Booth's Algorithm:

1. The multiplicand is subtracted from the partial product upon encountering the first least significand 1 in a string of I'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

Hardware Implementation of Booth Algorithm:

Figure 10-7 Hardware for Booth algorithm.



Note: Sign bit is not separated from register. QR register contains the multiplier register and $Q_n$represent the least significant bit of the multiplier in QR. Q_{n+1} is an extra flip flop appended to QR to facilitate a double bit inspection of the multiplier.
AC register and appended Q_{n+1} are initially cleared to 0.
Sequence counter Sc is set to the number n which is equal to the number of bits of bits In the multiplier.
Q_nQ_{n+1} are to successive bits in the multiplier

Example for multiplication using Boot h algorithm:

Q_nQ_{n+1}	BR = 1011 ,BR'+1 = 01001	AC	QR	Q_{n+1}	SC
10	Initial Subtract BR ashr	00000 01001 01001 00100	10011 11001	0 1	101 100
11	ashr	00010	01100	1	011
01	Add BR ashr	10111 11001 11100	 10110	 0	 010
00	ashr	11110	01011	0	001
10	Subtract BR	01001			

	Ashr	00111 00011	10101	1	000

Algorithm in flowchart for multiplication of signed 2's complement numbers.

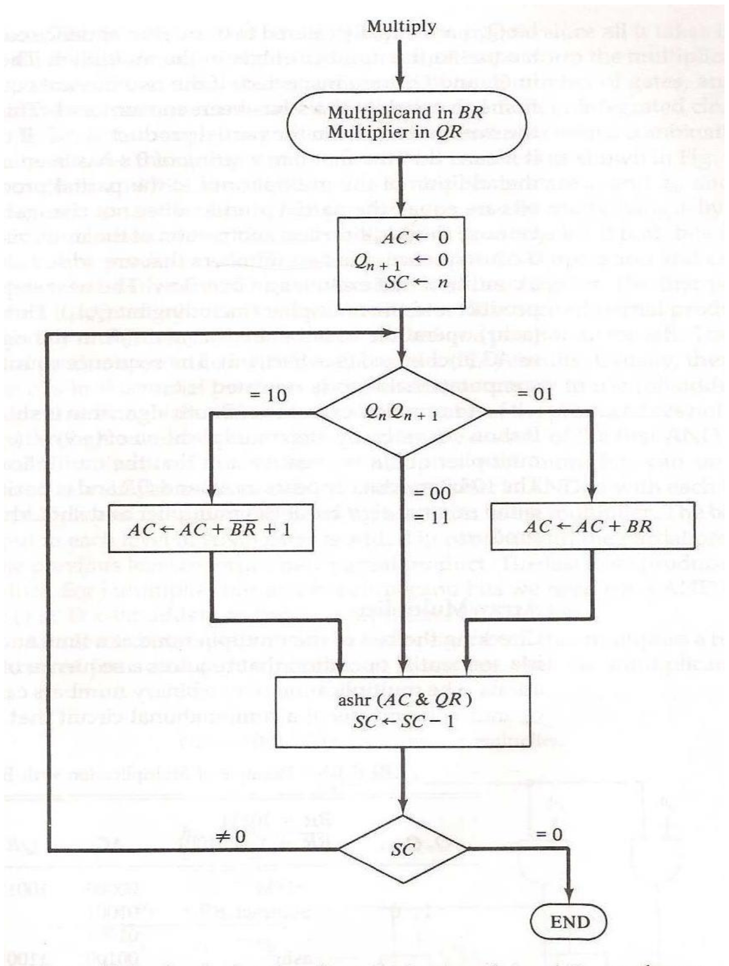


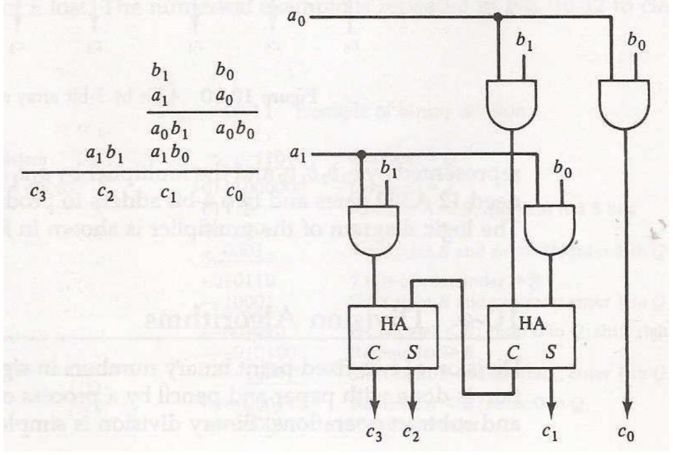
Figure 10-8 Booth algorithm for multiplication of signed-2's complement numbers.

Array Multiplier:

2 -bit by 2- bit Array Multiplier:

Multiplicand bits are b_1 and b_0 .Multiplier bits are a_1 and a_0 .The first partial product is obtained by multiplying a_0 by b_1 b_0 . The bit multiplication is implemented by AND gate. First partial product is made by two AND gates. Second partial product is made by two AND gates. The two partial products are added with two half adder circuits.

Figure 10-9 2-bit by 2-bit array multiplier.



Combinational circuit binary multiplier:

A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there bits in the multiplier. The binary output in each level of the AND gates is added in parallel with the partial product of the previous level to form a ne partial product. The last level produces the product. For j multiplier and k multiplicand bits, we need j*k AND Gates and (j-1)*k bit adders to ptoduce a product of j+k bits.

4- bit by 3-bit Array Multiplier:

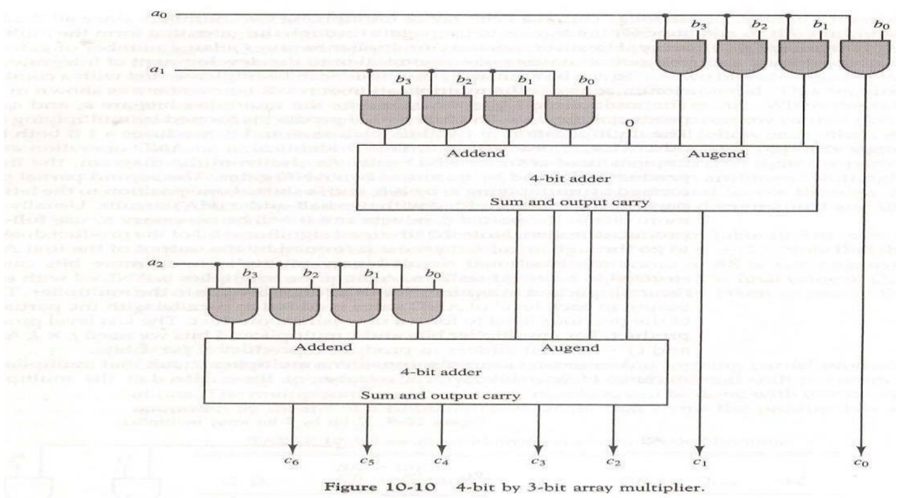


Figure 10-10 4-bit by 3-bit array multiplier.

<u>Division Algorithms:</u>

Division Process for division of fixed point binary number in signed –magnitude representation:

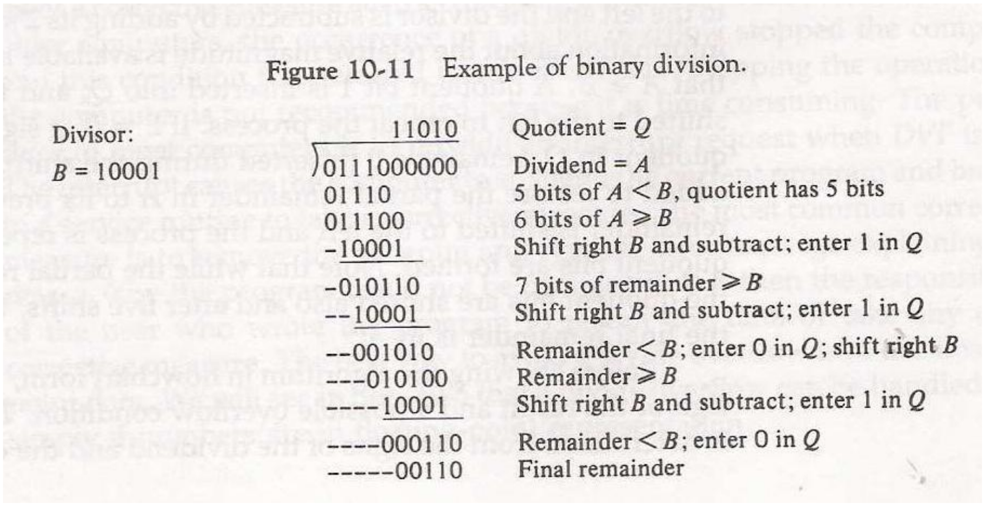


Figure 10-11 Example of binary division.

```
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
```

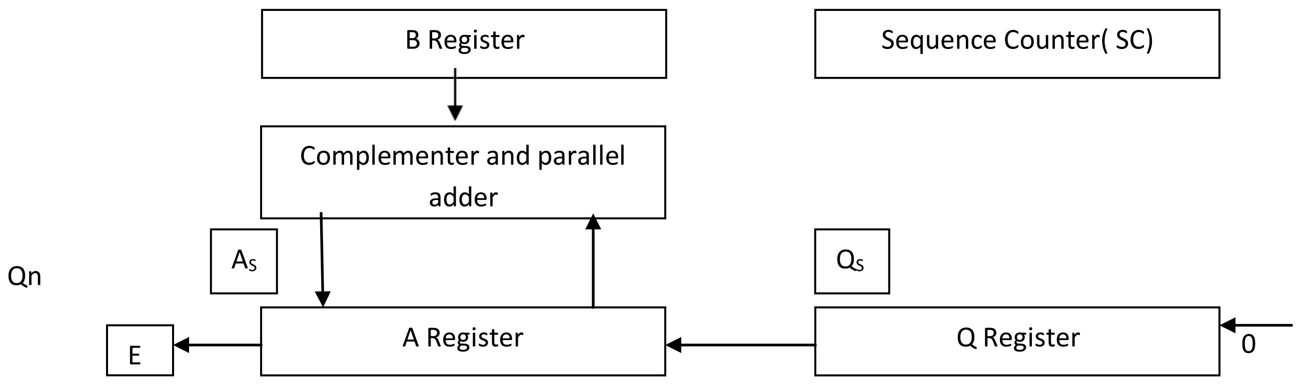
Let dividend A consists of 10 bits and divisor B consists of 5 bits.

1. Compare the 5 most significant bits of the dividend with that of divisor.
2. If the 5 bit number is smaller than divisor B, then take 6 bits of the dividend and compare with the 5 bit divisor.
3. The 6 bit number is greater than divisor B. Hence place a 1 for the quotient bit in the sixth position above the dividend. Shift the divisor once to the right and subtracted from the dividend. The difference is called partial remainder.
4. Repeat the process with the partial remainder and divisor. If the partial remainder is equal or 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 small than the divisor, then the quotient bit is zero and no subtraction is needed. The divisor is shifted once to the right in any case,.

Hardware Implementation of division for signed magnitude fixed point numbers:

To implement division using a digital computer, the process is changed slightly for convenience.

1. Instead of shifting the divisor to the right, the dividend or the partial remainder, is shifted to the left so as to leave the two numbers in the required relative position.
2. Subtraction may be achieved by adding A (dividend)to the 2's complement of B(divisor). The information about the relative magnitude is then available from end carry.
3. Register EAQ is now shifted to the left with 0 inserted into Q_n and the previous value of E is lost..
4. The divisor is stored in B register and the double length dividend is stored in registers A and Q.
5. The dividend is shifted to the left and the divisor is subtracted by adding it's 2's complement value.
6. If E= 1, it signifies that A ≥ B.A quotient bit is inserted into $Q_n$and the partial remainder is shifted to the left to repeat the process.
7. If E = 0, it signifies that A < B so the quotient 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 all 5 quotient bits are formed.
8. At the end Q contains the quotient and A the remainder. If the sign of dividend and divisor are alike, the quotient is positive and if unalike, it is negative. The sign of the remainder is the same as dividend.



Hardware for implementing division of fixed point signed- Magnitude Numbers

Example of Binary division with digital hardware: Divisor B = 10001, $\overline{B}$ + 1 = 01111

		E	**A**	**Q**	**SC**
	Dividend:		01110	00000	5
	Shl EAQ		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 , **B + 1**		01111		
	E = 1	1	00101		
	Set Q_n= 1	1	00101	00011	3
	Shl EAQ	0	01010	00110	
	Add , **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 , **B + 1**		01111 ___________		
	E = 1	1	00011		
	Set Q_n= 1	1	00011	01101	1
	Shl EAQ	0	00110	11010	
	Add , **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	11010	
	Quotient in Q				

Divide 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 higher order half bits of the dividend constitute a number greater than or equal to the divisor. If the divisor is zero, then the dividend will definitely be greater than or equal to divisor. Hence divide overflow condition occurs and hence the divide-overflow –flip flop will be set. Let the flip flop be called DVF.

Handling DVF:

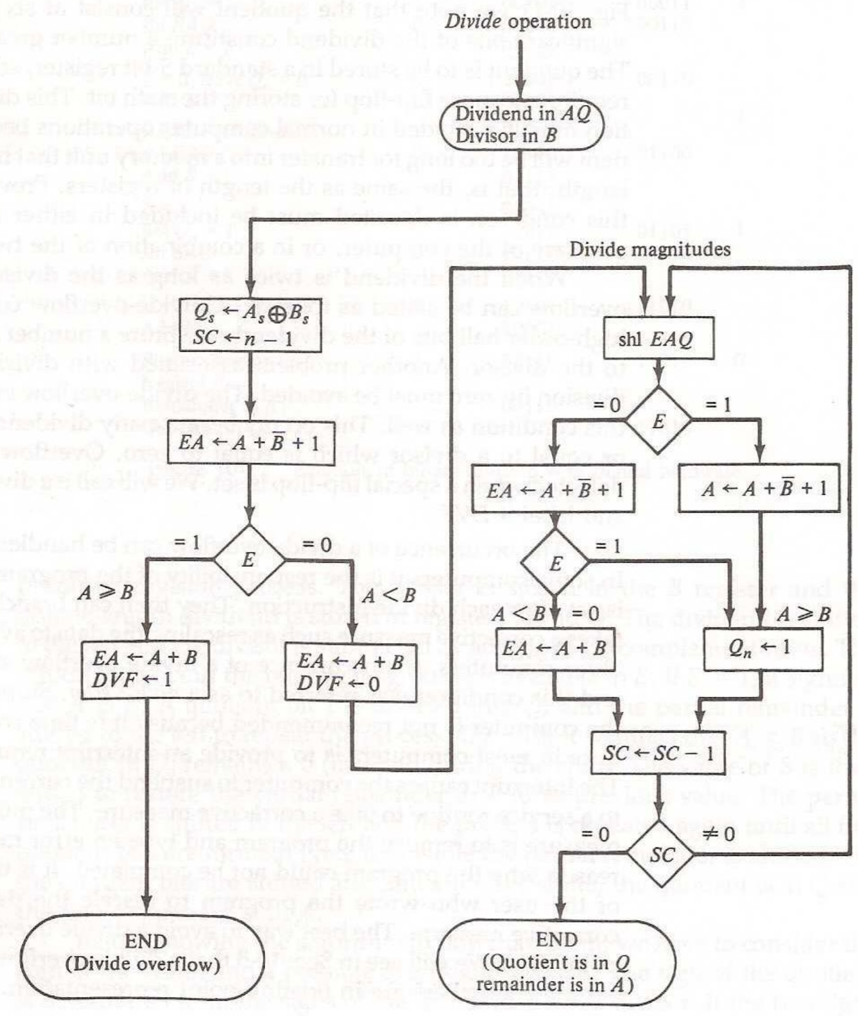
1. Check if DVF is set after each divide instruction. If DVF is set, then the program branches to a subroutine that takes corrective measures such as rescaling the data to avoid overflow.
2. An interrupt is generated if DVF is set. The interrupt causes the processor to suspend the current program and branch to interrupt service routine to take corrective measure. The most

common corrective measure is to remove the program and type an error message that explains the reasons.

3. The divide overflow can be handled very simply if the numbers are represented in floating point representation.

Flow chart for divide operation:

Figure 10-13 Flowchart for divide operation.



Assumption:

Operands are transferred from memory to registers as n bit words.n-1 bit form magnitude and 1 bit shows the sign.

A divide overflow condition is tested by subtracting the divisor in B from half of the bits of dividend stored in A. If vA ≥ B, the DVF is set and the operation is terminated prematurely. If A < B, no DVF occurs and so the value of 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 higher order bit shifted into E. If the bit shifted into E is 1, we know that EA is greater than 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. Since register A is missing the higher order bit of the dividend (which is in E), it's value is $EA - 2^{n-1}$. Adding to this value the 2's complement of B results in

$(EA-2^{n-1}) + (2^{n-1} - B) = E-B$. The carry from the addition is not transferred to E if we want E to remain a 1.

If the shift left operation inserts a zero into E, the divisor is subtracted by adding it's 2's complement value and the carry is transferred into E. If E = 1, it signifies that A ≥ B and hence Q_n 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 Q_n .(0 was inserted during the shift).

This process is repeated again with register A holding the partial remainder. After n-1 times, the quotient magnitude is formed in the register Q and the remainder is found in register A.

10-5 Floating-Point Arithmetic Operations

Many high-level programming languages have a facility for specifying floating-point numbers. 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 operations are quite often included in the internal hardware. If no hardware is available for the operations, the compiler must be designed with a package of floating-point software subroutines. Although the hardware method is more expensive, it is so much more efficient than the software method that floating-point hardware is included in most computers and is omitted only in very small ones.

integer declaration statement

Basic Considerations

Floating-point representation of data was introduced in Sec. 3-4. 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; 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. For example, assume a fraction representation and a radix 10. The decimal number 537.25 is represented in a register with $m = 53725$ and $e = 3$ and is interpreted to represent the floating-point number

$$.53725 \times 10^3$$

A floating-point number is 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 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. 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 $\pm(2^{47} - 1)$, which is approximately $\pm 10^{14}$. 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 accommodated is

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

This number is derived from a fraction that contains 35 1's, an exponent of 11 bits (excluding its sign), and the fact that $2^{11} - 1 = 2047$. The largest number that can be accommodated is approximately 10^{615}, which is a very large number. The mantissa can accommodate 35 bits (excluding the sign) and if considered as an integer it can store a number as large as $(2^{35} - 1)$. This is approximately equal to 10^{10}, which is equivalent to a decimal number of 10 digits.

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

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:

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

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. The usual alignment procedure is to shift the mantissa that has

2.

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:

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

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. In the example above, it is necessary to shift left twice to obtain $.35000 \times 10^3$. 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 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 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 compare and increment (for aligning the mantissas), add and subtract (for multiplication and division), and decrement (to normalize the result). 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. 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 from 49 to 00. The subtraction of 50 gives the negative values in the range of −1 to −50.

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. 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.

In the examples above, we used decimal numbers to demonstrate some of the concepts that must be understood when dealing with floating-point numbers. Obviously, the same concepts apply to binary numbers as well. The algorithms developed in this section are for binary numbers. Decimal computer arithmetic is discussed in the next section.

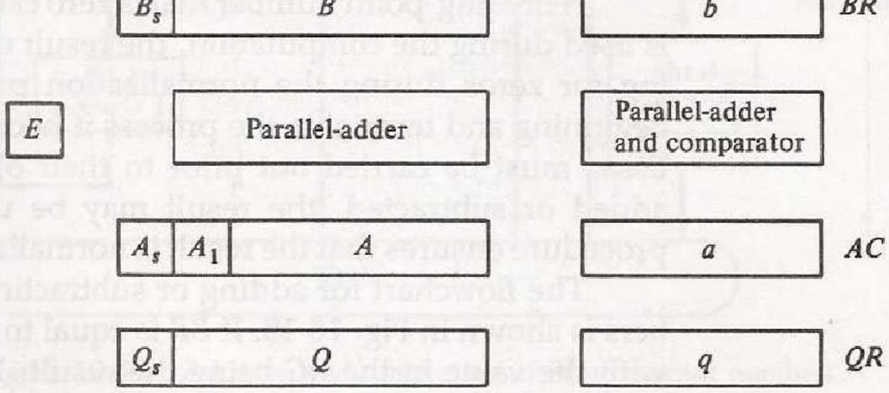
Register Configuration

The register configuration for floating-point operations is quite similar to the layout for fixed-point operations. As a general 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. 10-14. 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. The exponent part uses the corresponding lowercase letter symbol.

It is assumed that each floating-point number has a mantissa in signed-magnitude representation and a biased exponent. Thus the *AC* has a mantissa

Figure 10-14 Registers for floating-point arithmetic operations.



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.

Similarly, register *BR* is subdivided into B_s, B, and b, and *QR* into Q_s, 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. It is assumed that the floating-point numbers are so large that the chance of an exponent overflow is very remote, and for this reason 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. 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 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 unnormalized. 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. 10-15. 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

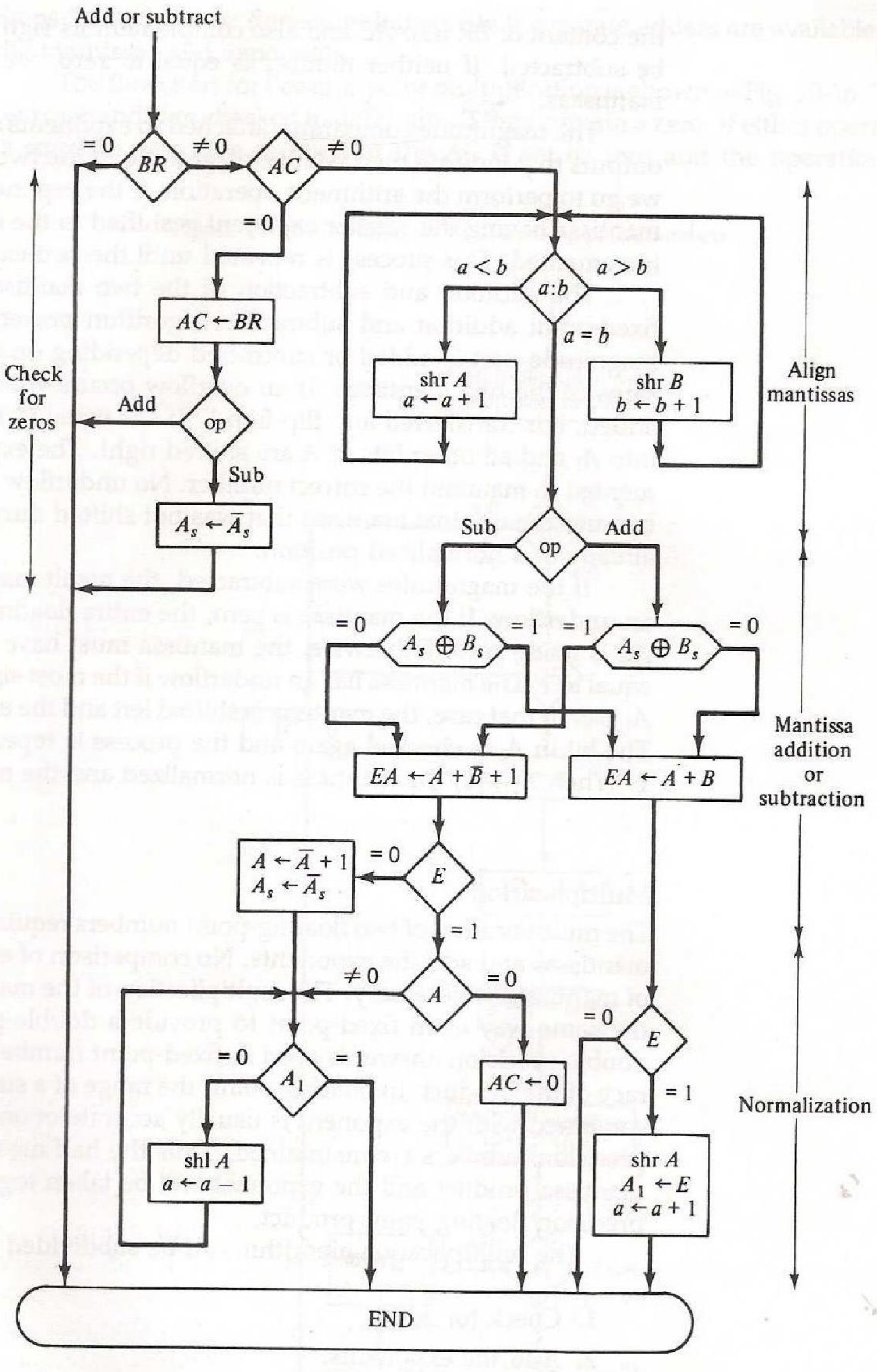


Figure 10-15 Addition and subtraction of floating-point numbers.

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 algorithm presented in Fig. 10-2. 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.

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 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.

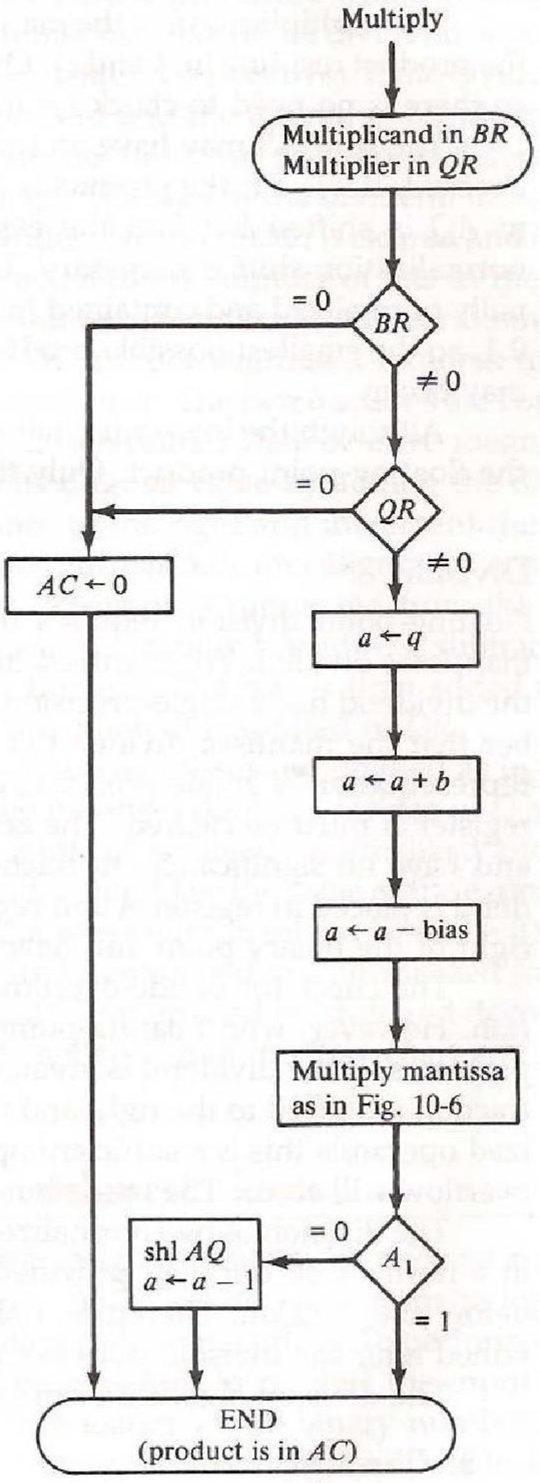
The multiplication algorithm can be subdivided into four parts:

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 flowchart for floating-point multiplication is shown in Fig. 10-16. 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

Figure 10-16 Multiplication of floating-point numbers.



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 no 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. Remember that the mantissa dividend is a fraction and not an integer. 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.

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 divide-overflow will occur. The operation above is referred to as a *dividend alignment*.

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 a normalization.

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 flowchart for floating-point division is shown in Fig. 10-17. 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.

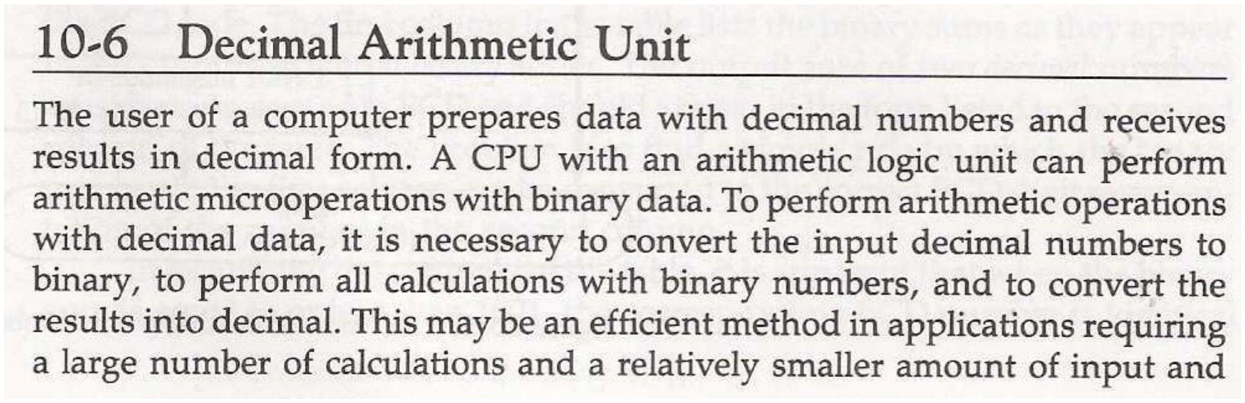
The dividend alignment is similar to the divide-overflow check in the fixed-point operation. The proper alignment requires that the fraction dividend be smaller than the divisor. The two fractions are compared by a subtraction test. The carry in E determines their relative magnitude. The dividend fraction 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 $A < B$.

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. 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. The floating-point quotient is already normalized and resides in QR. The exponent of the remainder should be the same as the exponent of the dividend. The binary point for the remainder mantissa lies $(n - 1)$ positions to the left of A_1. The remainder can be converted to a normalized fraction by subtracting $n - 1$ from the dividend exponent and by shift and decrement until the bit in A_1 is equal to 1. This is not shown in the flow chart and is left as an exercise.

lecture-10.docx

1.1.2.11 Lecture-11



10-6 Decimal Arithmetic Unit

The user of a computer prepares data with decimal numbers and receives results in decimal form. A CPU with an arithmetic logic unit can perform arithmetic microoperations with binary data. To perform arithmetic operations with decimal data, it is necessary to convert the input decimal numbers to binary, to perform all calculations with binary numbers, and to convert the results into decimal. This may be an efficient method in applications requiring a large number of calculations and a relatively smaller amount of input and

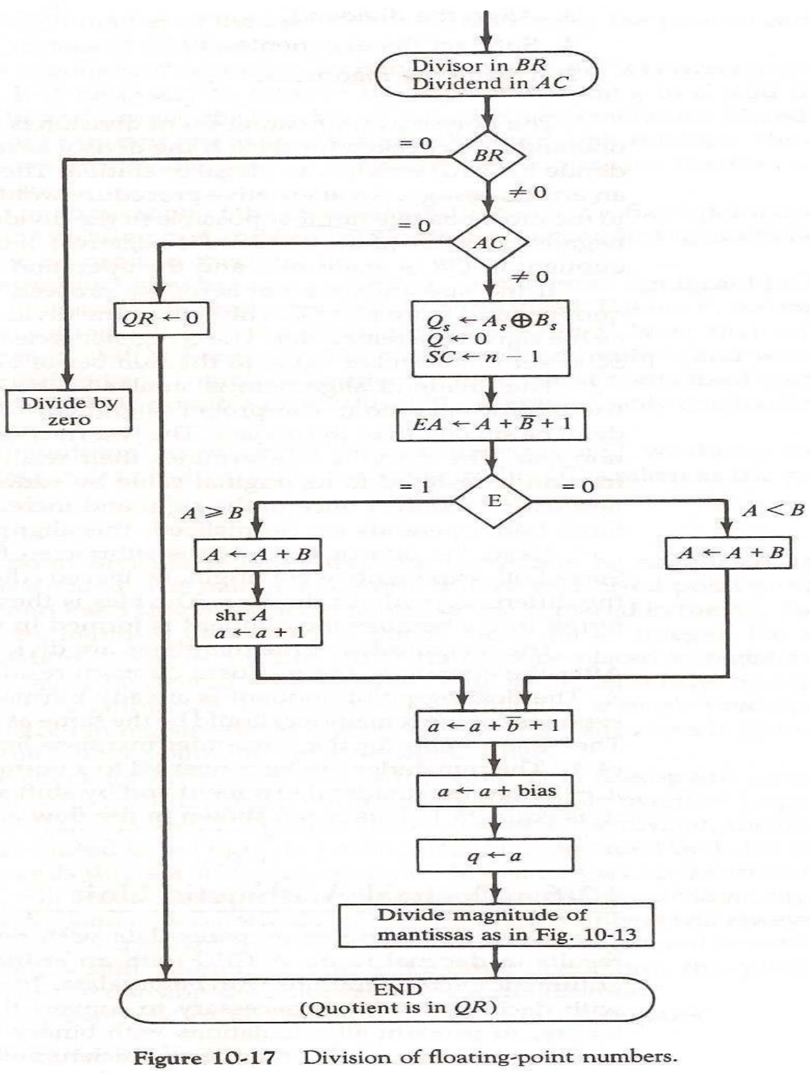


Figure 10-17 Division of floating-point numbers.

output data. When the application calls for a large amount of input–output and a relatively smaller number of arithmetic calculations, it becomes convenient to do the internal arithmetic directly with the decimal numbers. Computers capable of performing decimal arithmetic must store the decimal data in binary-coded form. The decimal numbers are then applied to a decimal arithmetic unit capable of executing decimal arithmetic microoperations.

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, since this process requires 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 perform calculations with binary or decimal data.

A decimal arithmetic unit is a digital function that performs decimal microoperations. It can add or subtract decimal numbers, usually by forming the 9's or 10's complement of the subtrahend. The unit accepts coded decimal numbers and generates results in the same adopted binary code. A single-stage decimal arithmetic unit consists 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 augend digit, four inputs for the addend digit, and an input-carry. The outputs include four terminals for the sum digit and one for the output-carry. Of course, there is a wide variety of possible circuit configurations dependent on the code used to represent the decimal digits.

BCD Adder

Consider the arithmetic addition of two decimal digits in BCD, together with a possible carry from a previous stage. Since each input digit does not exceed 9, the output sum cannot be greater than $9 + 9 + 1 = 19$, the 1 in the sum being an input-carry. Suppose that we apply two BCD digits to a 4-bit binary adder. The adder will form the sum in *binary* and produce a result that may range from 0 to 19. These binary numbers are listed in Table 10-4 and are labeled by symbols K, Z_8, Z_4, Z_2, and Z_1. K is the carry and the subscripts under the letter Z represent the weights 8, 4, 2, and 1 that can be assigned to the four bits in the BCD code. The first column in the table lists the binary sums as they appear in the outputs of a 4-bit *binary* adder. The output sum of two *decimal* numbers must be represented in BCD and should appear in the form listed in the second column of the table. The problem is to find a simple rule by which the binary number in the first column can be converted to the correct BCD digit representation of the number in the second column.

In examining the contents of the table, it is apparent that when the binary sum is equal to or less than 1001, the corresponding BCD number is identical

TABLE 10-4 Derivation of BCD Adder

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

and therefore no conversion is needed. When the binary sum is greater than 1001, we obtain a nonvalid BCD representation. The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output-carry as required.

One method of adding decimal numbers in BCD would be to employ one 4-bit binary adder and perform the arithmetic operation one digit at a time. The low-order pair of BCD digits is first added to produce a binary sum. If the result is equal or greater than 1010, it is corrected by adding 0110 to the binary sum. This second operation will automatically produce an output-carry for the next pair of significant digits. The next higher-order pair of digits, together with the input-carry, is then added to produce their binary sum. If this result is equal to or greater than 1010, it is corrected by adding 0110. The procedure is repeated until all decimal digits are added.

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 Z_8. To distinguish them from binary 1000 and 1001 which also have a 1 in position Z_8, we specify further that either Z_4

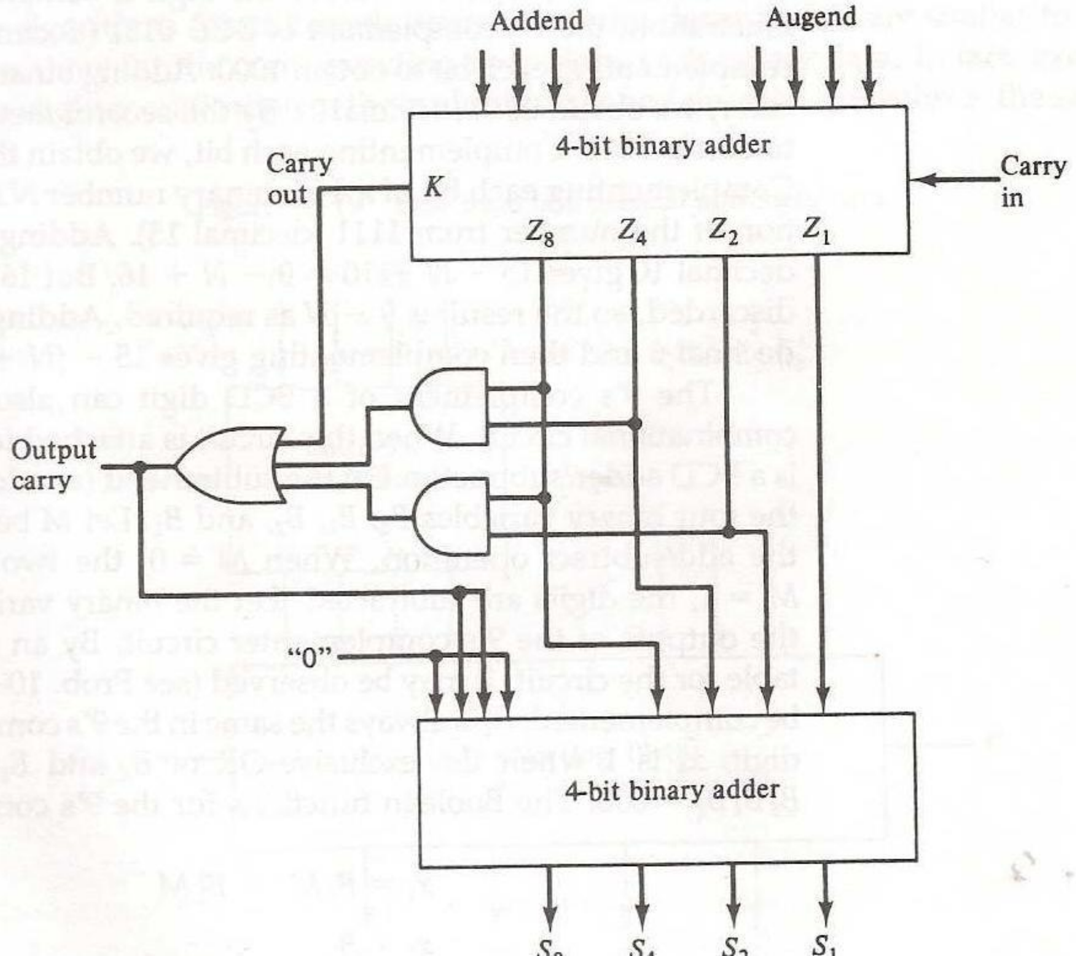
or Z_2 must have a 1. The condition for a correction and an output-carry can be expressed by the Boolean function

$$C = K + Z_8 Z_4 + Z_8 Z_2$$

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

A BCD adder is a circuit that adds two BCD digits in parallel and produces a sum digit also in BCD. A BCD adder must include the correction logic in its internal construction. To add 0110 to the binary sum, we use a second 4-bit binary adder as shown in Fig. 10-18. The two decimal digits, together with the input-carry, are first added in the top 4-bit binary adder to produce the binary sum. When the output-carry is equal to 0, nothing is added to the binary sum. When it is equal to 1, binary 0110 is added to the binary sum through the bottom 4-bit binary adder. The output-carry generated from the bottom binary adder may be ignored, since it supplies information already available in the output-carry terminal.

Figure 10-18 Block diagram of BCD adder.



A decimal parallel-adder that adds n decimal digits needs n BCD adder stages with the output-carry from one stage connected to the input-carry of the next-higher-order stage. To achieve shorter propagation delays, BCD adders include the necessary circuits for carry look-ahead. Furthermore, the adder circuit for the correction does not need all four full-adders, and this circuit can be optimized.

BCD Subtraction

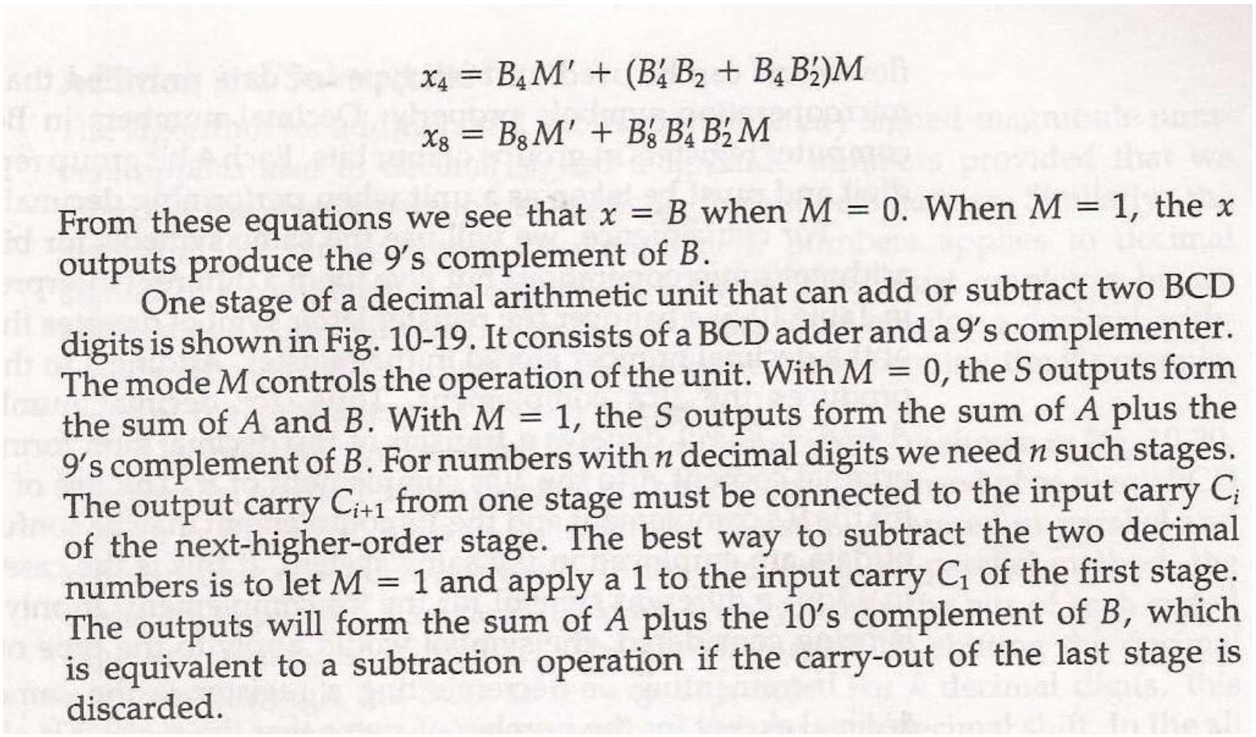
A straight subtraction of two decimal numbers will require a subtractor circuit that will be somewhat different from a BCD adder. It is more economical to perform the subtraction by taking the 9's or 10's complement of the subtrahend and adding it to the minuend. Since the BCD is not a self-complementing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

The 9's complement of a decimal digit represented in BCD may be obtained by complementing the bits in the coded representation of the digit provided a correction is included. There are two possible correction methods. In the first method, binary 1010 (decimal 10) is added to each complemented digit and the carry discarded after each addition. In the second method, binary 0110 (decimal 6) is added before the digit is complemented. As a numerical illustration, the 9's complement of BCD 0111 (decimal 7) is computed by first complementing each bit to obtain 1000. Adding binary 1010 and discarding the carry, we obtain 0010 (decimal 2). By the second method, we add 0110 to 0111 to obtain 1101. Complementing each bit, we obtain the required result of 0010. Complementing each bit of a 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 $15 - N + 10 = 9 - N + 16$. 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.

The 9's complement of a BCD digit can also be obtained through a combinational circuit. When this circuit is attached to a BCD adder, the result is a BCD adder/subtractor. Let the subtrahend (or addend) digit be denoted by the four binary variables B_8, B_4, B_2, and B_1. Let M be a mode bit that controls the add/subtract operation. When $M = 0$, the two digits are added; when $M = 1$, the digits are subtracted. Let the binary variables x_8, x_4, x_2, and x_1 be the outputs of the 9's complementer circuit. By an examination of the truth table for the circuit, it may be observed (see Prob. 10-30) that B_1 should always be complemented; B_2 is always the same in the 9's complement as in the original digit; x_4 is 1 when the exclusive-OR of B_2 and B_4 is 1; and x_8 is 1 when $B_8 B_4 B_2 = 000$. The Boolean functions for the 9's complementer circuit are

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

$$x_2 = B_2$$



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

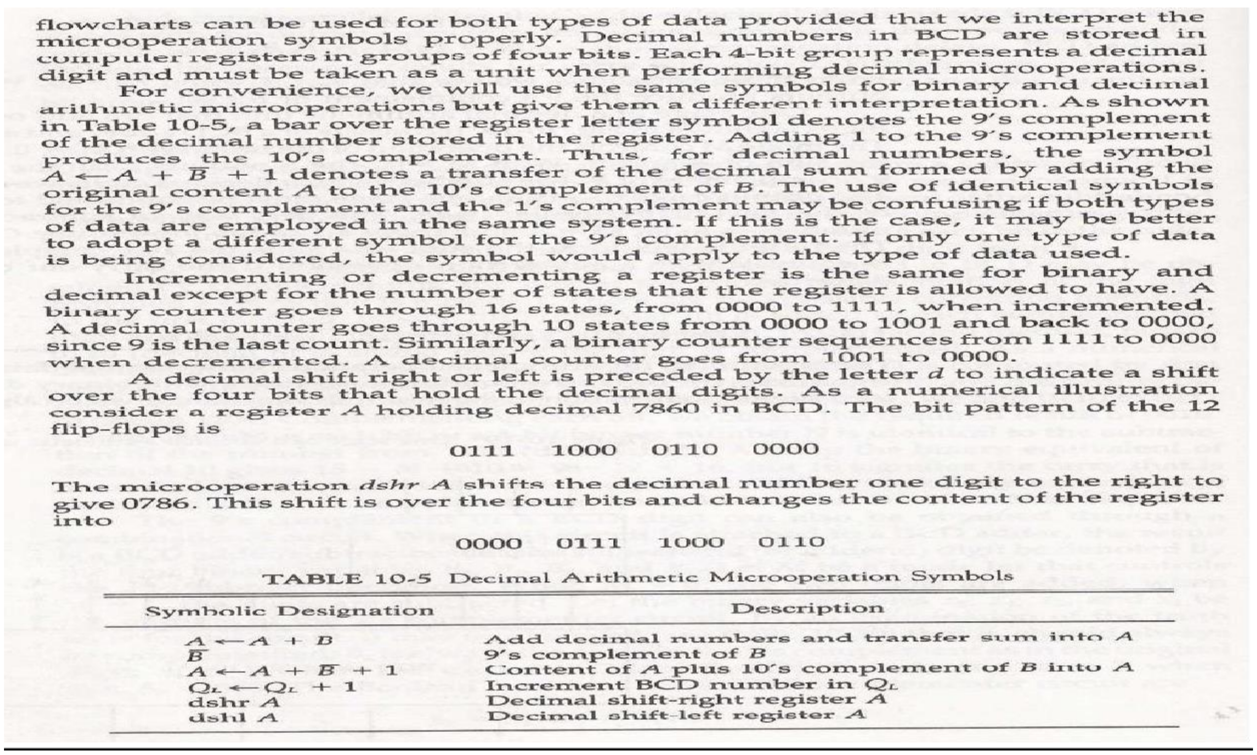
$$x_8 = B_8M' + B_8'B_4'B_2'M$$

From these equations we see that $x = B$ when $M = 0$. When $M = 1$, the x outputs produce the 9's complement of B.

One stage of a decimal arithmetic unit that can add or subtract two BCD digits is shown in Fig. 10-19. It consists of a BCD adder and a 9's complementer. The mode M controls the operation of the unit. With $M = 0$, the S outputs form the sum of A and B. With $M = 1$, the S outputs form the sum of A plus the 9's complement of B. For numbers with n decimal digits we need n such stages. The output carry C_{i+1} from one stage must be connected to the input carry C_i of the next-higher-order stage. The best way to subtract the two decimal numbers is to let $M = 1$ and apply a 1 to the input carry C_1 of the first stage. The outputs will form the sum of A plus the 10's complement of B, which is equivalent to a subtraction operation if the carry-out of the last stage is discarded.

lecture-11.docx

1.1.2.12 Lecture-12



flowcharts can be used for both types of data provided that we interpret the microoperation symbols properly. Decimal numbers in BCD are stored in computer registers in groups of four bits. Each 4-bit group represents a decimal digit and must be taken as a unit when performing decimal microoperations.

For convenience, we will use the same symbols for binary and decimal arithmetic microoperations but give them a different interpretation. As shown in Table 10-5, a bar over the register letter symbol denotes the 9's complement of the decimal number stored in the register. Adding 1 to the 9's complement produces the 10's complement. Thus, for decimal numbers, the symbol $A \leftarrow A + \overline{B} + 1$ denotes a transfer of the decimal sum formed by adding the original content A to the 10's complement of B. The use of identical symbols for the 9's complement and the 1's complement may be confusing if both types of data are employed in the same system. If this is the case, it may be better to adopt a different symbol for the 9's complement. If only one type of data is being considered, the symbol would apply to the type of data used.

Incrementing or decrementing a register is the same for binary and decimal except for the number of states that the register is allowed to have. A binary counter goes through 16 states, from 0000 to 1111, when incremented. A decimal counter goes through 10 states from 0000 to 1001 and back to 0000, since 9 is the last count. Similarly, a binary counter sequences from 1111 to 0000 when decremented. A decimal counter goes from 1001 to 0000.

A decimal shift right or left is preceded by the letter d to indicate a shift over the four bits that hold the decimal digits. As a numerical illustration consider a register A holding decimal 7860 in BCD. The bit pattern of the 12 flip-flops is

0111 1000 0110 0000

The microoperation *dshr A* shifts the decimal number one digit to the right to give 0786. This shift is over the four bits and changes the content of the register into

0000 0111 1000 0110

TABLE 10-5 Decimal Arithmetic Microoperation Symbols

Symbolic Designation	Description
$A \leftarrow A + B$	Add decimal numbers and transfer sum into A
$\overline{B}$	9's complement of B
$A \leftarrow A + \overline{B} + 1$	Content of A plus 10's complement of B into A
$Q_L \leftarrow Q_L + 1$	Increment BCD number in Q_L
dshr A	Decimal shift-right register A
dshl A	Decimal shift-left register A

Addition and Subtraction

The algorithm for addition and subtraction of binary signed-magnitude numbers applies also to decimal signed-magnitude numbers provided that we interpret the microoperation symbols in the proper manner. Similarly, the algorithm for binary signed-2's complement numbers applies to decimal signed-10's complement numbers. The binary data must employ a binary adder and a complementer. The decimal data must employ a decimal arithmetic unit capable of adding two BCD numbers and forming the 9's complement of the subtrahend as shown in Fig. 10-19.

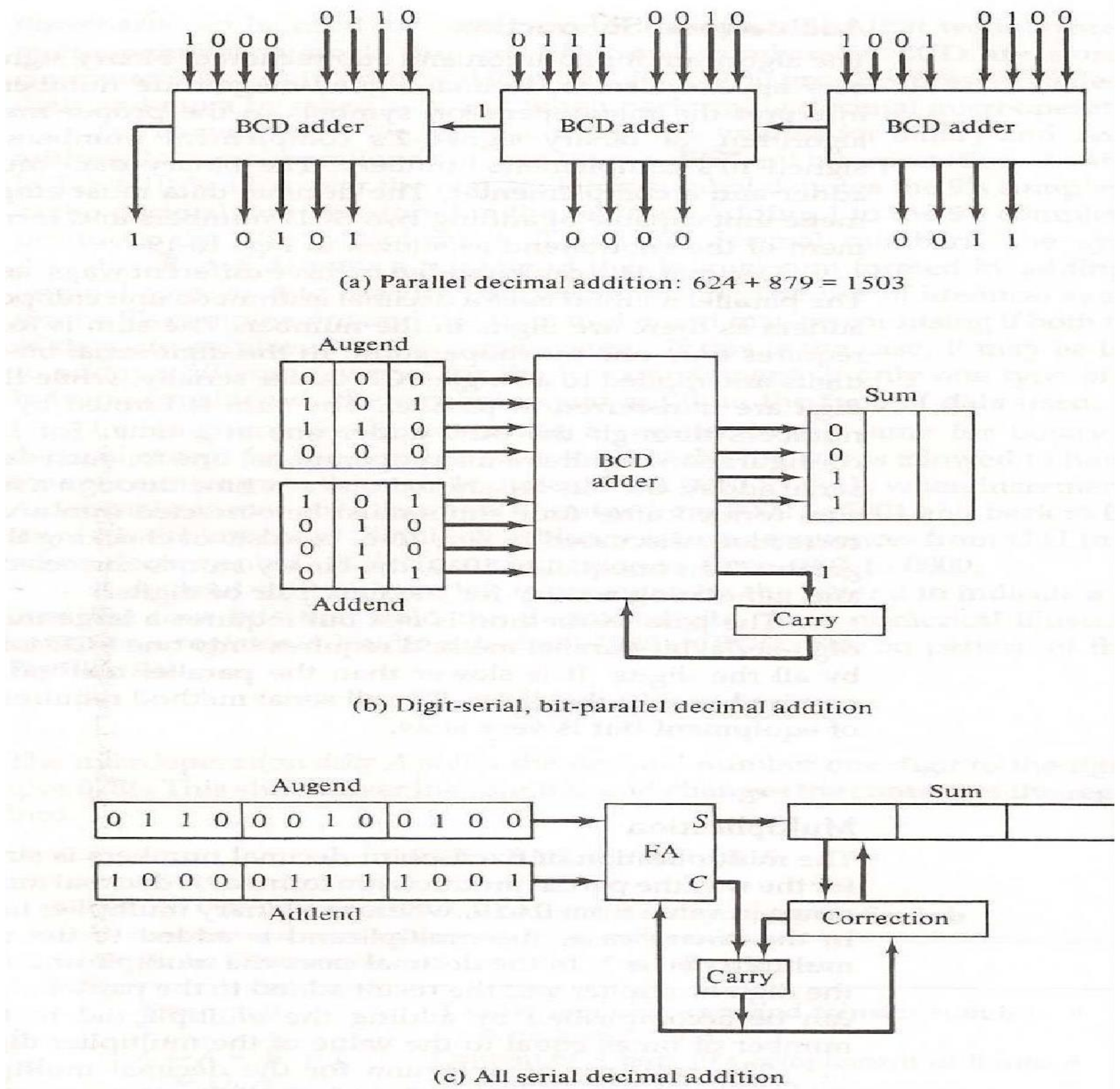
Decimal data can be added in three different ways, as shown in Fig. 10-20. The parallel method uses a decimal arithmetic unit composed of as many BCD adders as there are digits in the number. The sum is formed in parallel and requires only one microoperation. In the digit-serial bit-parallel method, the digits are applied to a single BCD adder serially, while the bits of each coded digit are transferred in parallel. The sum is formed by shifting the decimal numbers through the BCD adder one at a time. For k decimal digits, this configuration requires k microoperations, one for each decimal shift. In the all serial adder, the bits are shifted one at a time through a full-adder. The binary sum formed after four shifts must be corrected into a valid BCD digit. This correction, discussed in Sec. 10-6, consists of checking the binary sum. If it is greater than or equal to 1010, the binary sum is corrected by adding to it 0110 and generating a carry for the next pair of digits.

The parallel method is fast but requires a large number of adders. The digit-serial bit-parallel method requires only one BCD adder, which is shared by all the digits. It is slower than the parallel method because of the time required to shift the digits. The all serial method requires a minimum amount of equipment but is very slow.

Multiplication

The multiplication of fixed-point decimal numbers is similar to binary except for the way the partial products are formed. A decimal multiplier has digits that range in value from 0 to 9, whereas a binary multiplier has only 0 and 1 digits. In the binary case, the multiplicand is added to the partial product if the multiplier bit is 1. In the decimal case, the multiplicand must be multiplied by the digit multiplier and the result added to the partial product. This operation can be accomplished by adding the multiplicand to the partial product a number of times equal to the value of the multiplier digit.

The registers organization for the decimal multiplication is shown in Fig. 10-21. We are assuming here four-digit numbers, with each digit occupying four bits, for a total of 16 bits for each number. There are three registers, A, B, and Q, each having a corresponding sign flip-flop A_s, B_s, and Q_s.



(a) Parallel decimal addition: 624 + 879 = 1503

(b) Digit-serial, bit-parallel decimal addition

(c) All serial decimal addition

Figure 10-20 Three ways of adding decimal numbers.

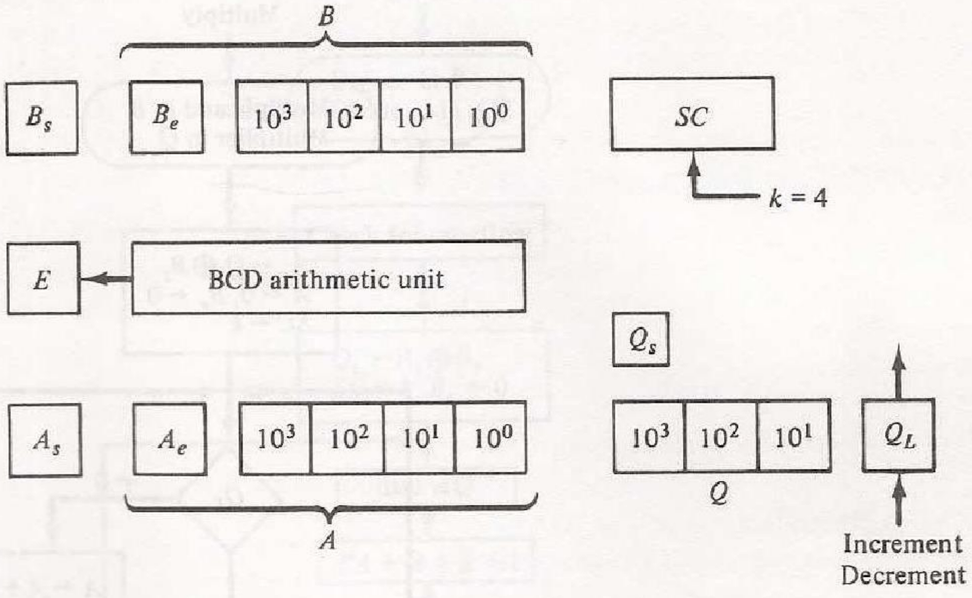


Figure 10-21 Registers for decimal arithmetic multiplication and division.

Registers A and B have four more bits designated by A_e and B_e that provide an extension of one more digit to the registers. The BCD arithmetic unit adds the five digits in parallel and places the sum in the five-digit A register. The end-carry goes to flip-flop E. The purpose of digit A_e is to accommodate an overflow while adding the multiplicand to the partial product during multiplication. The purpose of digit B_e is to form the 9's complement of the divisor when subtracted from the partial remainder during the division operation. The least significant digit in register Q is denoted by Q_L. This digit can be incremented or decremented.

A decimal operand coming from memory consists of 17 bits. One bit (the sign) is transferred to B_s and the magnitude of the operand is placed in the lower 16 bits of B. Both B_e and A_e are cleared initially. The result of the operation is also 17 bits long and does not use the A_e part of the A register.

The decimal multiplication algorithm is shown in Fig. 10-22. Initially, the entire A register and B_e are cleared and the sequence counter SC is set to a number k equal to the number of digits in the multiplier. The low-order digit of the multiplier in Q_L is checked. If it is not equal to 0, the multiplicand in B is added to the partial product in A once and Q_L is decremented. Q_L is checked again and the process is repeated until it is equal to 0. In this way, the multiplicand in B is added to the partial product a number of times equal to the multiplier digit. Any temporary overflow digit will reside in A_e and can range in value from 0 to 9.

Next, the partial product and the multiplier are shifted once to the right. This places zero in A_e and transfers the next multiplier quotient into Q_L. The process is then repeated k times to form a double-length product in AQ.

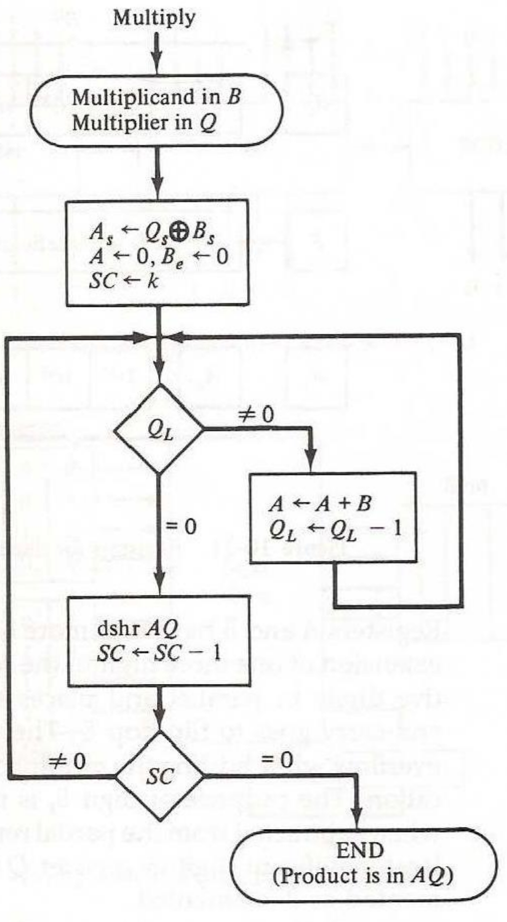


Figure 10-22 Flowchart for decimal multiplication.

Division

Decimal division is similar to binary division except of course that the quotient digits may have any of the 10 values from 0 to 9. In the restoring division method, the divisor is subtracted from the dividend or partial remainder as many times as necessary until a negative remainder results. The correct remainder is then restored by adding the divisor. The digit in the quotient reflects the number of subtractions up to but excluding the one that caused the negative difference.

The decimal division algorithm is shown in Fig. 10-23. It is similar to the algorithm with binary data except for the way the quotient bits are formed. The dividend (or partial remainder) is shifted to the left, with its most significant digit placed in A_e. The divisor is then subtracted by adding its 10's complement value. Since B_e is initially cleared, its complement value is 9 as required. The carry in E determines the relative magnitude of A and B. If $E = 0$, it signifies

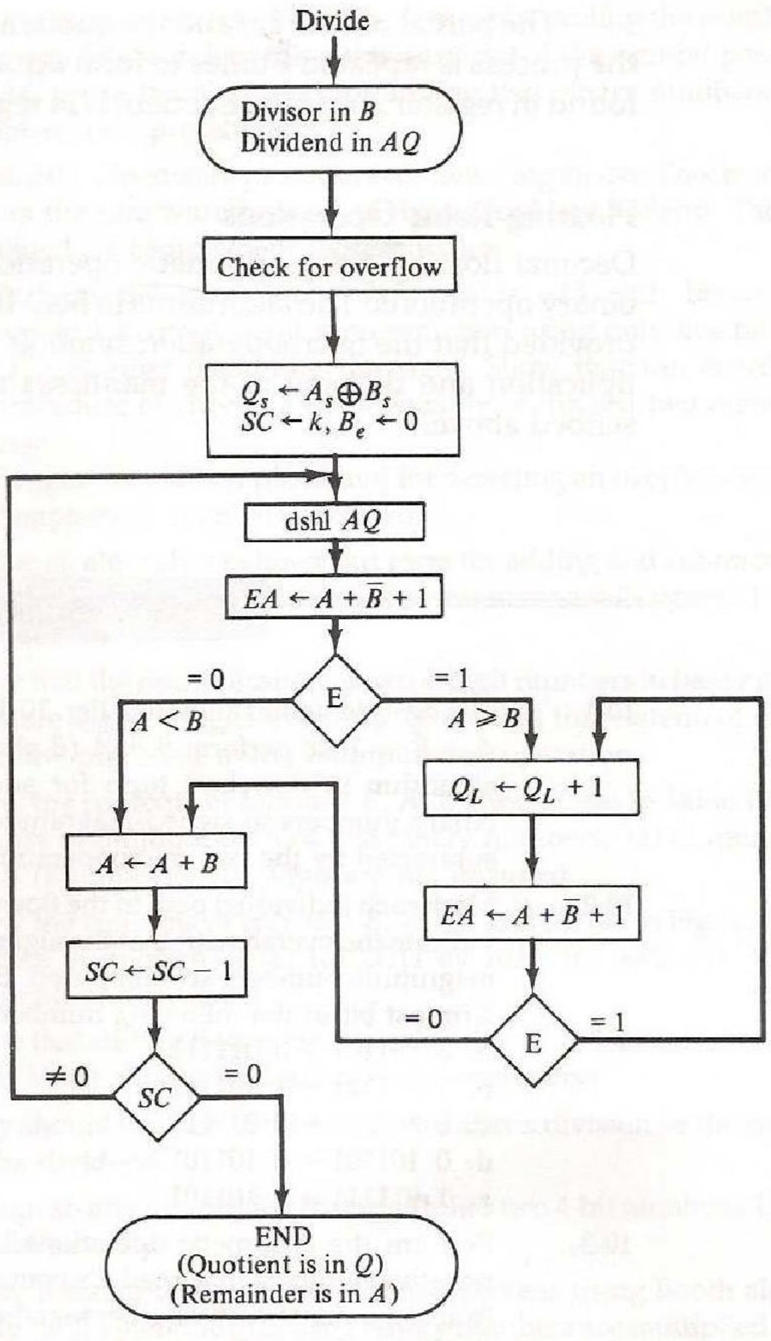
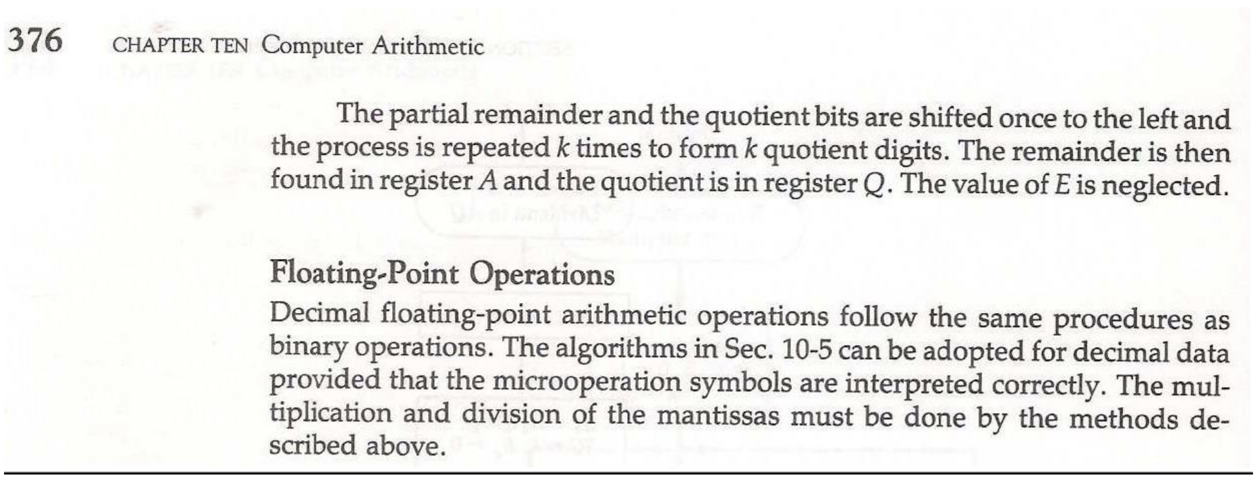


Figure 10-23 Flowchart for decimal division.

that $A < B$. In this case the divisor is added to restore the partial remainder and Q_L stays at 0 (inserted there during the shift). If $E = 1$, it signifies that $A \geq B$. The quotient digit in Q_L is incremented once and the divisor subtracted again. This process is repeated until the subtraction results in a negative difference which is recognized by E being 0. When this occurs, the quotient digit is not incremented but the divisor is added to restore the positive remainder. In this way, the quotient digit is made equal to the number of times that the partial remainder "goes" into the divisor.



376 CHAPTER TEN Computer Arithmetic

The partial remainder and the quotient bits are shifted once to the left and the process is repeated *k* times to form *k* quotient digits. The remainder is then found in register *A* and the quotient is in register *Q*. The value of *E* is neglected.

Floating-Point Operations

Decimal floating-point arithmetic operations follow the same procedures as binary operations. The algorithms in Sec. 10-5 can be adopted for decimal data provided that the microoperation symbols are interpreted correctly. The multiplication and division of the mantissas must be done by the methods described above.

Fill in the blanks type of questions <Minimum of ten>

a. The decimal representation for hex number F3 is__________________.

b. The binary equivalent for the decimal number 41.6875 is ______________________

c. The BCD code for the decimal number 248 is ____________________________.

d. For a given number N in base r having n digits, the (r-1)'s complement of N is defined as ______________

e. The 10's complement of a decimal number is obtained by adding ___ to the 9's complement value.

f. When 2 unsigned numbers are added, an overflow is detected from the ________________ of the most significant position.

g. An overflow for addition/ subtraction of two signed numbers is detected when the carry into the sign bit position and carry out of the sign bit position are__________________.

h. Booth multiplication algorithm is followed when the binary integers are represented in ______________

i. When Booth algorithm is used for multiplication, the partial product does not change when the multiplier .is identical to the previous multiplier ___.

j. Floating point multiplication and division do not require an alignment of the ______________ .

Answers: (1). 243 (2) 101001.1011 (3) 0010 0100 1000

(4) (r^n-1)-N (5) 1 (6) carry out (7) not equal

(8) signed 2's complement representation for negative integers. (9) bit, bit

(10) mantissa

Multiple choice questions<Minimum of ten>

1. Floating point representation is used to store
(A) Boolean values (B) whole numbers (C) real integers (D) integers
Ans: C

2. In computers, subtraction is generally carried out by
(A) 9's complement (B) 10's complement (C) 1's complement (D) 2's complement

Ans: D

3. The circuit used to store one bit of data is known as
(A) Register (B) Encoder (C) Decoder (D) Flip Flop
Ans: D

4. Which of the following is not a weighted code?
(A) Decimal Number system (B) Excess 3-cod
(C) Binary number System (D) None of these
Ans: B

5. Assembly language
(A) uses alphabetic codes in place of binary numbers used in machine language
(B) is the easiest language to write programs
(C) need not be translated into machine language

(D) None of these
Ans: A

6. The multiplicand register & multiplier register of a hardware circuit implementing booth's algorithm have (11101) & (1100). The result shall be
(A) (812) 10 (B) (-12) 10 (C) (12) 10 (D) (-812) 10
Ans: A

7. What characteristic of RAM memory makes it not suitable for permanent storage?
(A) too slow (B) unreliable (C) it is volatile (D) too bulky
Ans: C

8. (2FAOC) 16 is equivalent to
(A) (195 084) 10 (B) (001011111010 0000 1100) 2 (C) Both (A) and (B) (D) None of these
Ans: B

9. The average time required to reach a storage location in memory and obtain its contents is called the
(A) seek time (B) turnaround time (C) access time (D) transfer time
Ans: C

10. In signed-magnitude binary division, if the dividend is (11100) 2 and divisor is (10011) 2 then the result is
(A) (00100) 2 (B) (10100) 2 (C) (11001) 2 (D) (01100) 2
k. **Ans:** B

True or False questions<Minimum of ten>

Fill the blank with true or false.

1. EEPROM comes under volatile memory category. ____________
2. Thumb drive or pen drive is semiconductor memory. ____________
3. The control unit generates the appropriate signal at the right moment. ____________
4. While executing a program, CPU brings instruction and data from disk memory. ____________
5. A memory module of capacity 16 * 4 , indicates a storage of 128 bits. ____________
6. A memory module of capacity of 1024 locations, the required address bus size is 10. ____________
7. The program counter PC is used to store the address of the next instruction to be fetched from Accumulator. ____________

.

8. For n-bit signed integer, the range of numbers that can be represented is -2^{n-1} to 2^{n+1}. ________

9. Given a number N in base r having n digits, the (r-1)'s complement of N is defined as

   $(r^n - 1) - r$. ________

10. Floating point representation uses mantissa and an exponent part of radix R . ________

Answers: (1). false (2) true (3) true (4) false (5) false
(6) true (7) false (8) false (9)false (10) true