KTU NOTES

The learning companion.

KTU STUDY MATERIALS | SYLLABUS | LIVE NOTIFICATIONS | SOLVED QUESTION PAPERS

MODULE 5

Addition and Subtraction with Signed-2's Complement Data

The signed-2's complement representation of numbers together with arithmetic algorithms for addition and subtraction are introduced in Sec. 3-3. They are summarized here for easy reference. The leftmost bit of a binary number represents the sign bit: 0 for positive and 1 for negative. If the sign bit is 1, the entire number is represented in 2's complement form. Thus +33 is represented as 00100001 and −33 as 11011111. Note that 11011111 is the 2's complement of 00100001, and vice versa.

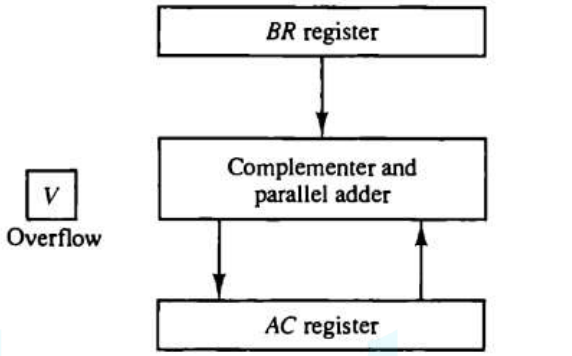
The addition of two numbers in signed-2's complement form consists of adding the numbers with the sign bits treated the same as the other bits of the number. A carry-out of the sign-bit position is discarded. The subtraction consists of first taking the 2's complement of the subtrahend and then adding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

When two numbers of n digits each are added and the sum occupies $n + 1$ digits, we say that an overflow occurred. The effect of an overflow on the sum of two signed-2's complement numbers is discussed in Sec. 3-3. An overflow can be detected by inspecting the last two carries out of the addition. When the two carries are applied to an exclusive-OR gate, the overflow is detected when the output of the gate is equal to 1.

The register configuration for the hardware implementation is shown in Fig. 10-3. This is the same configuration as in Fig. 10-1 except that the sign bits are not separated from the rest of the registers. 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 Fig. 10-4. The sum is obtained by adding the contents of *AC* and *BR* (including their sign bits). The overflow bit *V* is set to 1 if the exclusive-OR of the last two carries is 1, and it is cleared to 0 otherwise. The subtraction operation is accomplished by adding the content of *AC* to the 2's complement of *BR*. Taking the 2's complement of *BR* has the effect of changing a positive number to negative, and vice versa. An overflow must be checked during this operation because the two numbers added could have the same sign. The programmer must realize that if an overflow occurs, there will be an erroneous result in the *AC* register.

Figure 10-3 Hardware for signed-2's complement addition and subtraction.



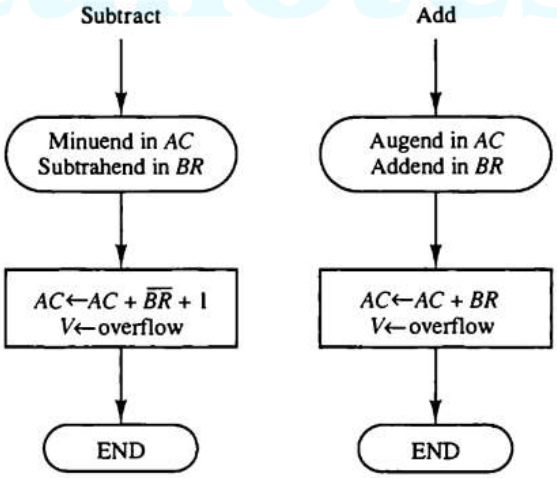


Figure 10-4 Algorithm for adding and subtracting numbers in signed-2's complement representation.

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. For this reason most computers adopt this representation over the more familiar signed-magnitude.

Floating Point Numbers

$$\pm M \times B^{E}$$

$$\pm \text{mantissa} \times \text{Base}^{\pm \text{Exponent}}$$

+ve or -ve (while) number with a decimal pt

↓

floating pt dec can float to any position.

eg 9×10^{8}

mantissa = 9

Base = 10

exp = 8

eg. 110×2^{7}

mantisse = 110

Base = 2

exp = 7

eg. 4364.784

$= 4364784 \times 10^{-3}$

Mantissa = 4364784

Base = 10

exp = -3.

Normalisation rules.

1. Integer part should be zero.

$$0.\ d_1\ d_2 \ldots\ d_n \times B^{\pm E}$$

$d_1 \neq 0$ & all $d_i^n \geq 0$.

eg. 1.23×10^{3}

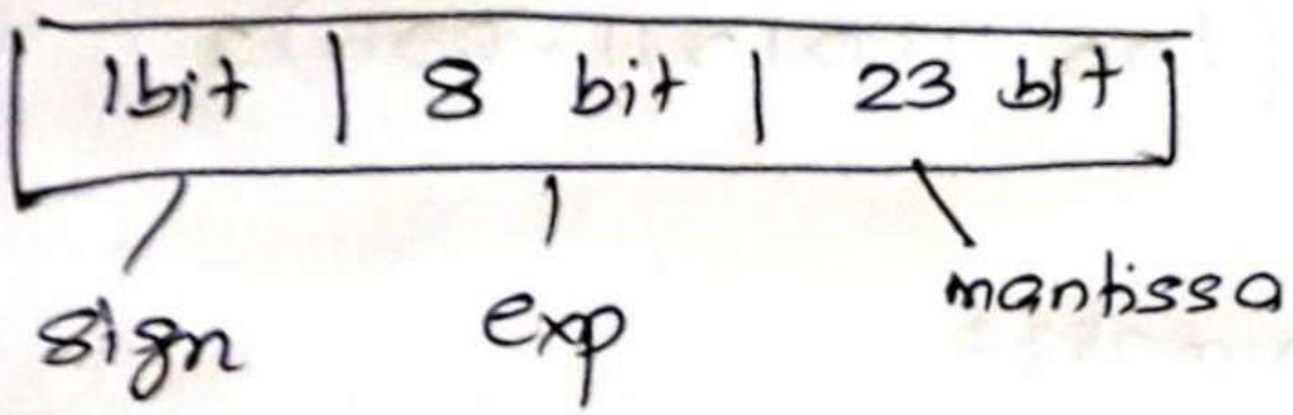
$\Rightarrow 0.123 \times 10^{4}$

$1.23 \times 10^{3} \times$

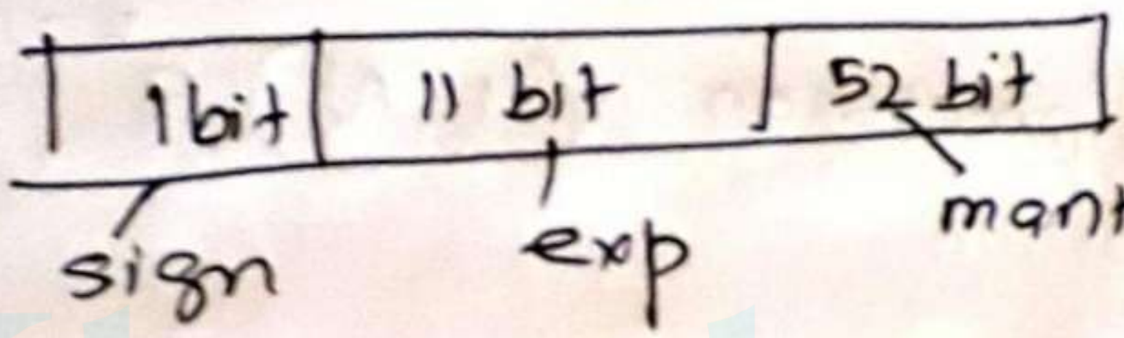
$0.0123 \times 10^{5} \times$

Floating point Representation

1. Single precision format (32 bit)



2. Double precision format (64 bit)



Steps.

1. Convert decimal to binary
2. Normalize the number
3. Apply single & double precision equations.

Single precision $(1.N)\,2^{E-127}$

Double precision $(1.N)\,2^{E-1023}$

Represent $(1259.125)_{10}$ in single & double precision format

1. $(100\ 1110\ 1011.001)_2$

2(1259, 1
2(629, 1
2(314, 0
2(157, 1
2(78, 0
2(39, 1
2(19, 1
2(9, 1
2(4, 0
2(2, 0
2(1, 1
0

$0.125 \times 2 =$
$0.25 \times 2 =$
$0.5 \times 2 =$

2. Normalize

$1.\ 00\ 111\ 010\ 1100\ 1 \times 2^{+10}$

3. Single precision $(1.N)\ 2^{E-127}$

$E - 127 = 10$

$E = 127 + 10 = 137.$

2(137, 1
2(68, 0
2(34, 0
2(17, 1
2(8, 0
2(4, 0
2(2, 0
2(1, 1
0

convert 137 → binary $= (10001001)_2$

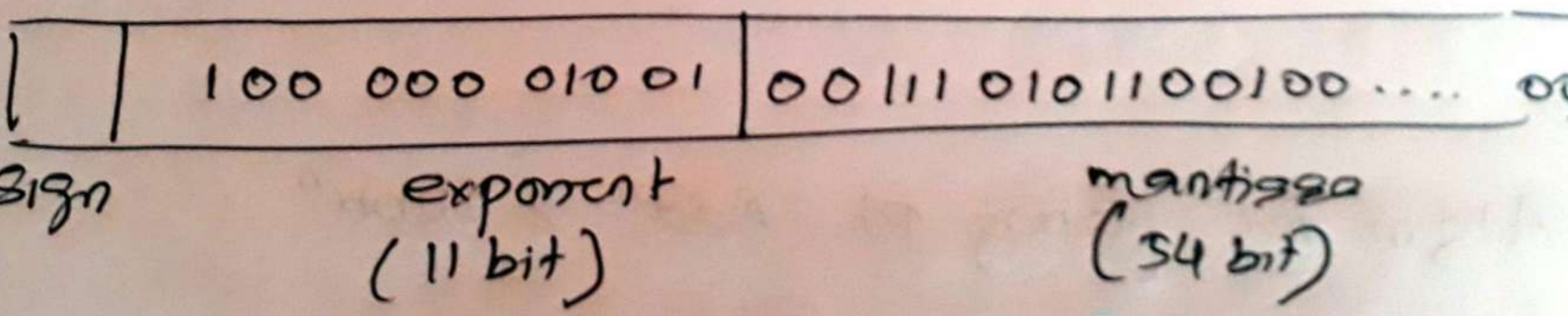
0	10001001	00111010110010000000000
1 bit	exponent 8 bit	mantissa 23 bit

Double precision $(1.N)\ 2^{E-1023}$

$E - 1023 = 10$

$E = 1033 \rightarrow$ binary

$= 1000\ 0001001$



eg Convert the decimal number 3.248×10^4
to IEEE 754 std. single precision floating pt. binary no.

$3.248 \times 10^4 = (32480)_{10}$

$= (1111110111100000)_2$

Normalizing $= 1.1111011110000 \times 2^{14}$

$S = 0$, $E = 14$, Biased $= 127 + E$

$= 127 + 14 = 141$

$= (10001101)$

$M = 1111011110000$

| 0 | 10001101 | 11111011110000000000000 |

eg. Find the decimal value corresponding to the given floating point number

(1, 10000010, 11110110000000000000000

$S = 1$

Biased E $= (10000010)_2$

$= (2^7 + 2^1) = 128 + 2 = (130)_{10}$

actual exponent, $E = 130 - 127$

$= 3$

Manhissa = 11110 11 ...

$= (2^{-1}) + (2^{-2}) + (2^{-3}) + (2^{-4}) + 0 + (2^{-6}) + (2$

$= 0.5 + 0.25 + 0.125 + 0.0625 + 0.015$

$+ 0.0078125$

$= (0.9609375)_{10}$

$(-1)^S \times (1 + M) \times 2^E$

$= (-1) \times (1.9609375) \times 2^3$

$= (-15.6875)_{10}$

eg2) A = 11001000 11000 1100010

1	10010001	10001100010000000000000

S = 1

biased E = $\underset{7\,6\,5\,4\,3\,2\,1\,0}{10010001}$

$= 2^7 + 2^4 + 2^0$

$= 128 + 16 + 1$

$= 145$

actual e = 145 - 127 = 18.

Mantissa = $\underset{-1\,-2\,-3\,-4\,-5\,-6\,-7\,-8\,-9\,-10}{1\,0\,0\,0\,1\,1\,0\,0\,0\,1}$

$= 2^{-1} + 2^{-5} + 2^{-6} + 2^{-10}$

$= 0.5 + 0.03125 + 0.015625 + 0.0009765$

$= 0.54787856$

Decimal No.

$= (-1)^S \times (1+m) \times 2^e$

$= (-1)^1 * (1+0.54787856) \times 2^{18}$

$= -1.5478 \times 2^{18}$

$= (-405746.483)_{10}$

eg3) A = 1/00010001/00010001 00000000 00000000

S = 1, biased E = $\underset{7\,6\,5\,4\,3\,2\,1\,0}{00010001}$ $= 2^4 + 2^0 = 17$ $e = 17 - 127 = -110$

Mantissa = $\underset{-1\,-2\,-3\,-4\,-5\,-6\,-7\,-8}{00010001}$ $= 2^{-4} + 2^{-8} = (0.06640625)_{10}$

$(-1)^S \times (1+m) \times 2^e = (-1)^1 \times (1.06640625) \times 2^{-110}$

$= -1.06640625 \times 2^{-110}$

eg Convert the decimal no. 3.248×10^4 to IEEE 754 Std. double precision floating pt. binary no.

$$3.248 \times 10^4 = (32480)_{10}$$
$$= (1111110111100000)_2$$

Normalizing $= 1.1111011110000 \times 2^{14}$

$S = 0$

$e = 14$

biased $E = 1023 + 14 = (1037)_{10}$
$= (1000\ 0000\ 1101)_2$

$M = 1111\ 0111\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000$

0	10000001101	1111011110000000000000000000000000000000000000000000
← 1 bit →	← 11 bits →	← 52 bits →

eg $(5EC.2CD)_{16}$

$(010111101100.001011001101)_2$

Normalizing

$= 1.01111011000010110011101 \times 2^{10}$

$S = 0, \quad E = 10$

biased exponent $E = 10 + 1023 = 1033$

$= (1000\ 0000\ 1001)_2$

$M = 0111\ 1011\ 0000\ 1011\ 0011\ 1010\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000$

0	10000001001	0111101100001011001101000000000000000000000000000000
1 bit	11 bits	52 bits

MODULE 5: Arithmetic Algorithms

Addition and Subtraction with Signed-Magnitude Data

The representation of numbers in signed-magnitude is familiar because it is used in everyday arithmetic calculations. The procedure for adding or subtracting two signed binary numbers with paper and pencil is simple and straightforward. A review of this procedure will be helpful for deriving the hardware algorithm.

We designate the magnitude of the two numbers by A and B. 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. These conditions are listed in the first column of Table 10-1. The other columns in the table show the actual operation to be performed with the *magnitude* of the numbers. The last column is needed to prevent a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not −0.

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

Addition (*subtraction*) *algorithm:* when the signs of A and B are identical (different), add the two magnitudes and attach the sign of A to the result. When the signs of A and B are different (identical), compare the magnitudes and

TABLE 10-1 Addition and Subtraction of Signed-Magnitude Numbers

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

subtract the smaller number from the larger. Choose the sign of the result to be the same as A if $A > B$ or the complement of the sign of A if $A < B$. If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

The two algorithms are similar except for the sign comparison. The procedure to be followed for identical signs in the addition algorithm is the same as for different signs in the subtraction algorithm, and vice versa.

Hardware Implementation

To implement the two arithmetic operations with hardware, it is first necessary that the two numbers be stored in registers. Let A and B be two registers that hold the magnitudes of the numbers, and A_s and B_s be two flip-flops that hold the corresponding signs. 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.

This procedure requires a magnitude comparator, an adder, and two subtractors. However, a different procedure can be found that requires less equipment. First, we know that subtraction can be accomplished by means of complement and add. Second, the result of a comparison can be determined from the end carry after the subtraction. Careful investigation of the alternatives reveals that the use of 2's complement for subtraction and comparison is an efficient procedure that requires only an adder and a complementer.

Figure 10-1 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. The A register provides other microoperations that may be needed when we specify the sequence of steps in the algorithm.

The addition of A plus B is done through the parallel adder. The S (sum) output of the adder is applied to the input of the A register. The complementer provides an output of B or the complement of B depending on the state of the mode control M. The complementer consists of exclusive-OR gates and the parallel adder consists of full-adder circuits as shown in Fig. 4-7 in Chap. 4. The M signal is also applied to the input carry of the adder. When $M = 0$, the output of B is transferred to the adder, the input carry is 0, and the output of

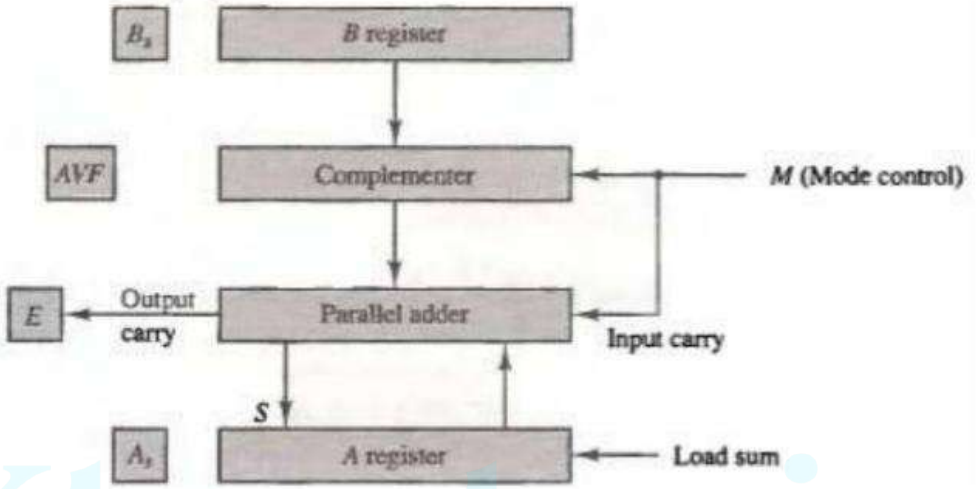


Figure 10-1 Hardware for signed-magnitude addition and subtraction.

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

Hardware Algorithm

The flowchart for the hardware algorithm is presented in Fig. 10-2. The two signs A_s and B_s are compared by an exclusive-OR gate. If the output of the gate is 0, the signs are identical; if it is 1, the signs are different. For an *add* operation, identical signs dictate that the magnitudes be added. For a *subtract* operation, different signs dictate that the magnitudes be added. The magnitudes are added with a microoperation $EA \leftarrow A + B$, where EA is a register that combines E and A. The carry in E after the addition constitutes an overflow if it is equal to 1. The value of E is transferred into the add-overflow flip-flop AVF.

The two magnitudes are subtracted if the signs are different for an *add* operation or identical for a *subtract* operation. The magnitudes are subtracted by adding A to the 2's complement of B. No overflow can occur if the numbers are subtracted so AVF is cleared to 0. A 1 in E indicates that $A \geq B$ and the number in A is the correct result. If this number 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. This operation can be done with one microoperation $A \leftarrow \bar{A} + 1$. However, we assume that the A register has circuits for microoperations *complement* and *increment*, so the 2's complement is obtained from these two microoperations. In other paths of the flowchart, the sign of the result is the same as the sign of A, so no change in A_s is required. However, when $A < B$, the sign of the result is the complement of the original sign of A. It is then necessary to complement A_s to obtain the correct sign. The final result is found in register A and its sign in A_s. The value in AVF provides an overflow indication. The final value of E is immaterial.

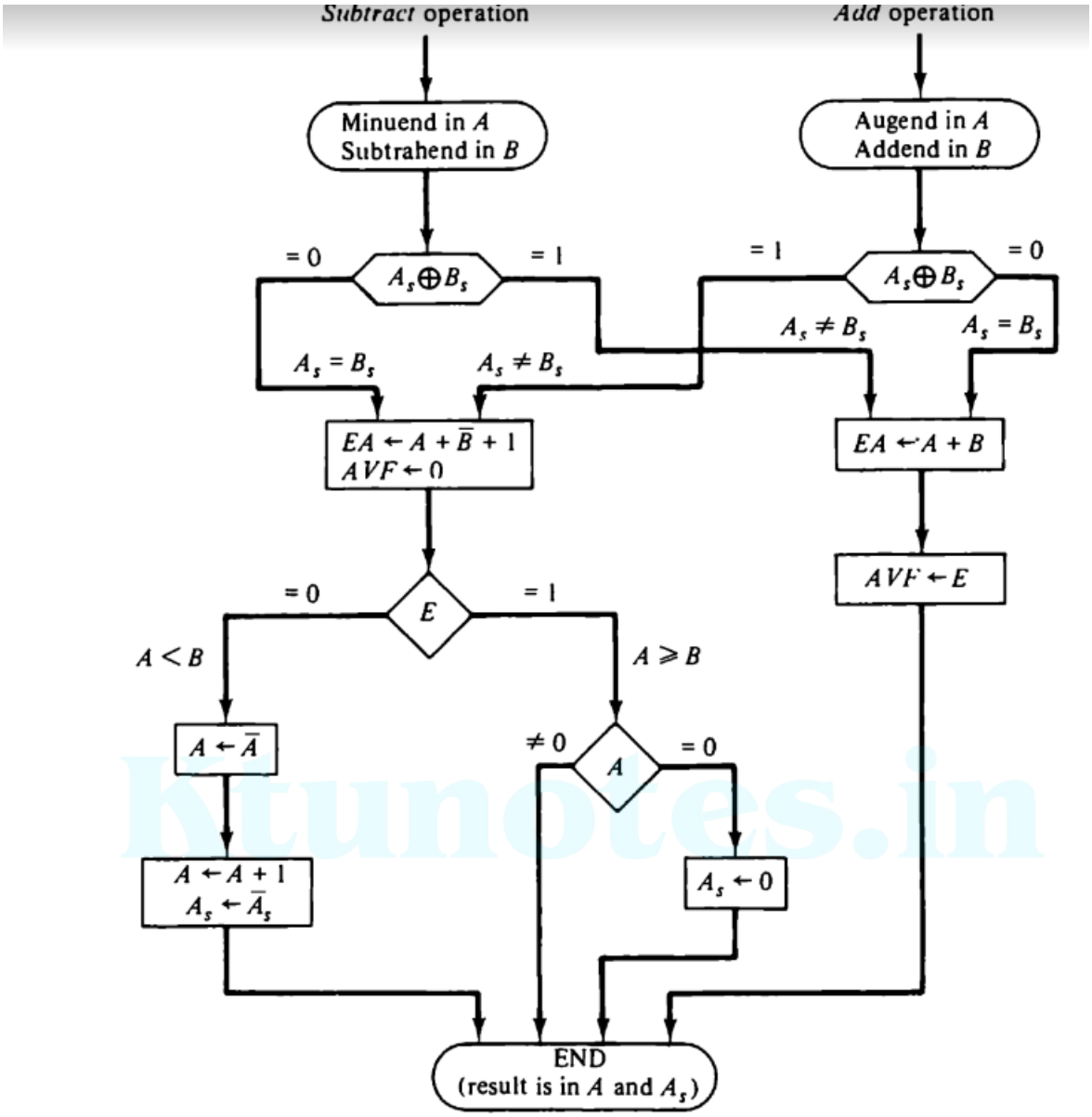


Figure 10-2 Flowchart for add and subtract operations.

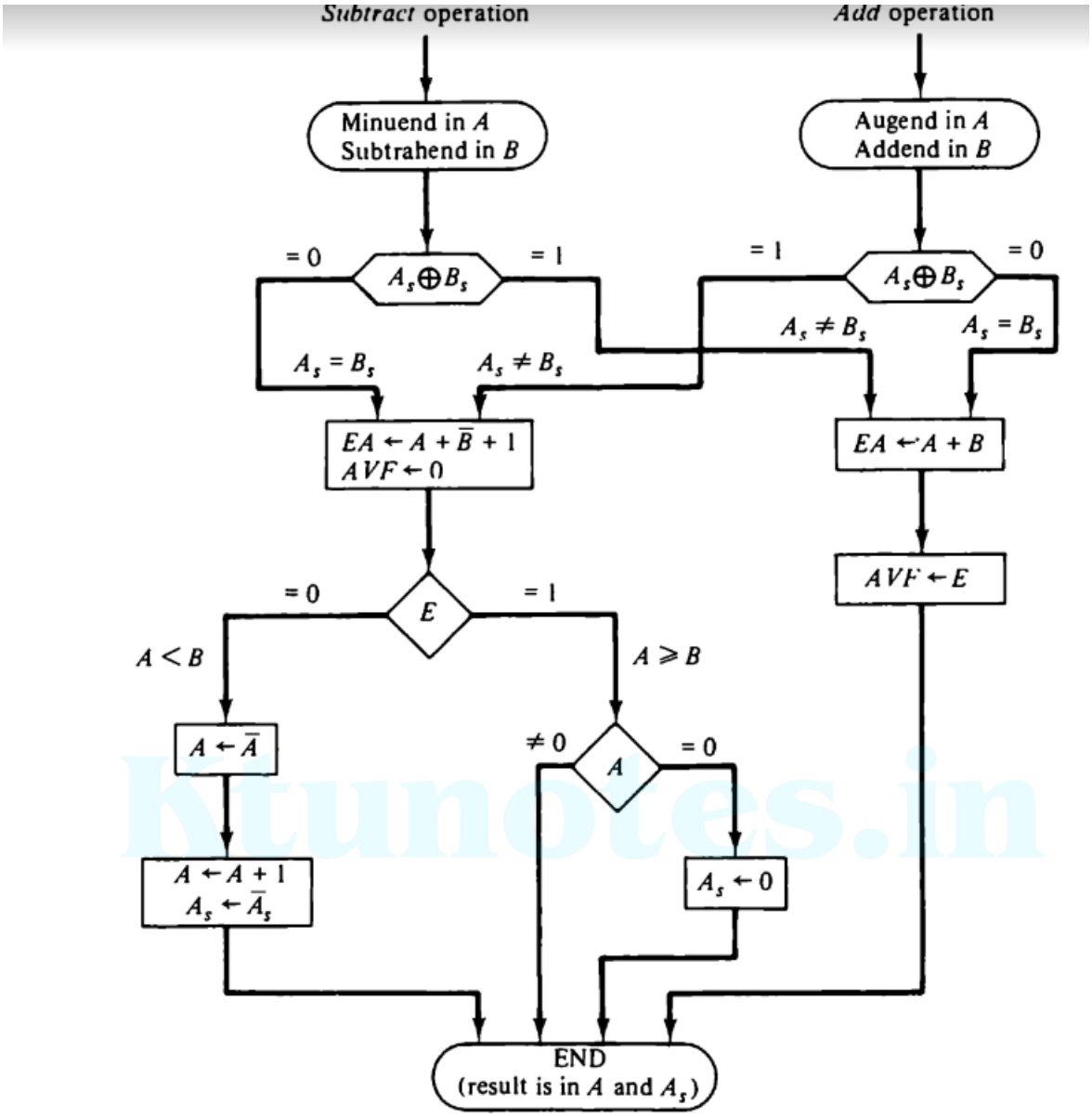


Figure 10-2 Flowchart for add and subtract operations.

Addition and Subtraction with Signed-2's Complement Data

The signed-2's complement representation of numbers together with arithmetic algorithms for addition and subtraction are introduced in Sec. 3-3. They are summarized here for easy reference. The leftmost bit of a binary number represents the sign bit: 0 for positive and 1 for negative. If the sign bit is 1, the entire number is represented in 2's complement form. Thus +33 is represented as 00100001 and −33 as 11011111. Note that 11011111 is the 2's complement of 00100001, and vice versa.

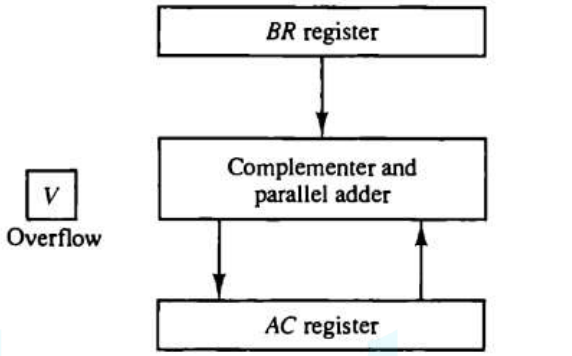
The addition of two numbers in signed-2's complement form consists of adding the numbers with the sign bits treated the same as the other bits of the number. A carry-out of the sign-bit position is discarded. The subtraction consists of first taking the 2's complement of the subtrahend and then adding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

When two numbers of n digits each are added and the sum occupies $n + 1$ digits, we say that an overflow occurred. The effect of an overflow on the sum of two signed-2's complement numbers is discussed in Sec. 3-3. An overflow can be detected by inspecting the last two carries out of the addition. When the two carries are applied to an exclusive-OR gate, the overflow is detected when the output of the gate is equal to 1.

The register configuration for the hardware implementation is shown in Fig. 10-3. This is the same configuration as in Fig. 10-1 except that the sign bits are not separated from the rest of the registers. 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 Fig. 10-4. The sum is obtained by adding the contents of *AC* and *BR* (including their sign bits). The overflow bit *V* is set to 1 if the exclusive-OR of the last two carries is 1, and it is cleared to 0 otherwise. The subtraction operation is accomplished by adding the content of *AC* to the 2's complement of *BR*. Taking the 2's complement of *BR* has the effect of changing a positive number to negative, and vice versa. An overflow must be checked during this operation because the two numbers added could have the same sign. The programmer must realize that if an overflow occurs, there will be an erroneous result in the *AC* register.

Figure 10-3 Hardware for signed-2's complement addition and subtraction.



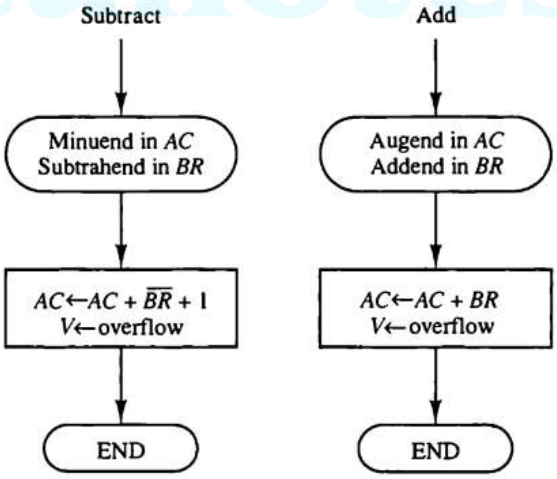


Figure 10-4 Algorithm for adding and subtracting numbers in signed-2's complement representation.

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. For this reason most computers adopt this representation over the more familiar signed-magnitude.

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.

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:

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

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:

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

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.

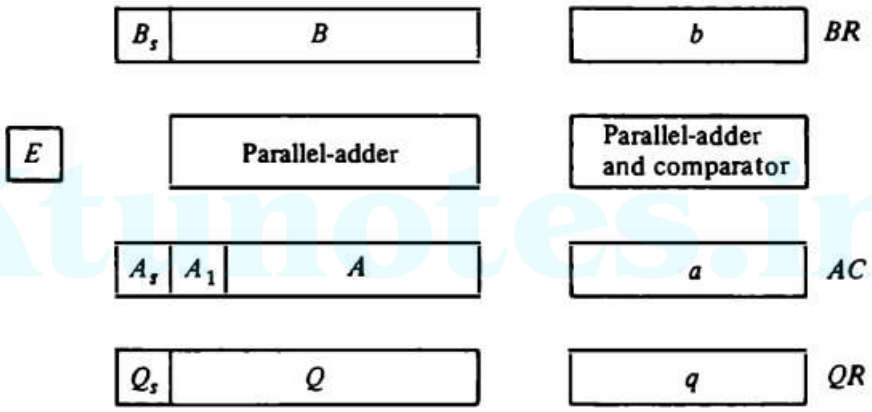
Register Configuration

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.

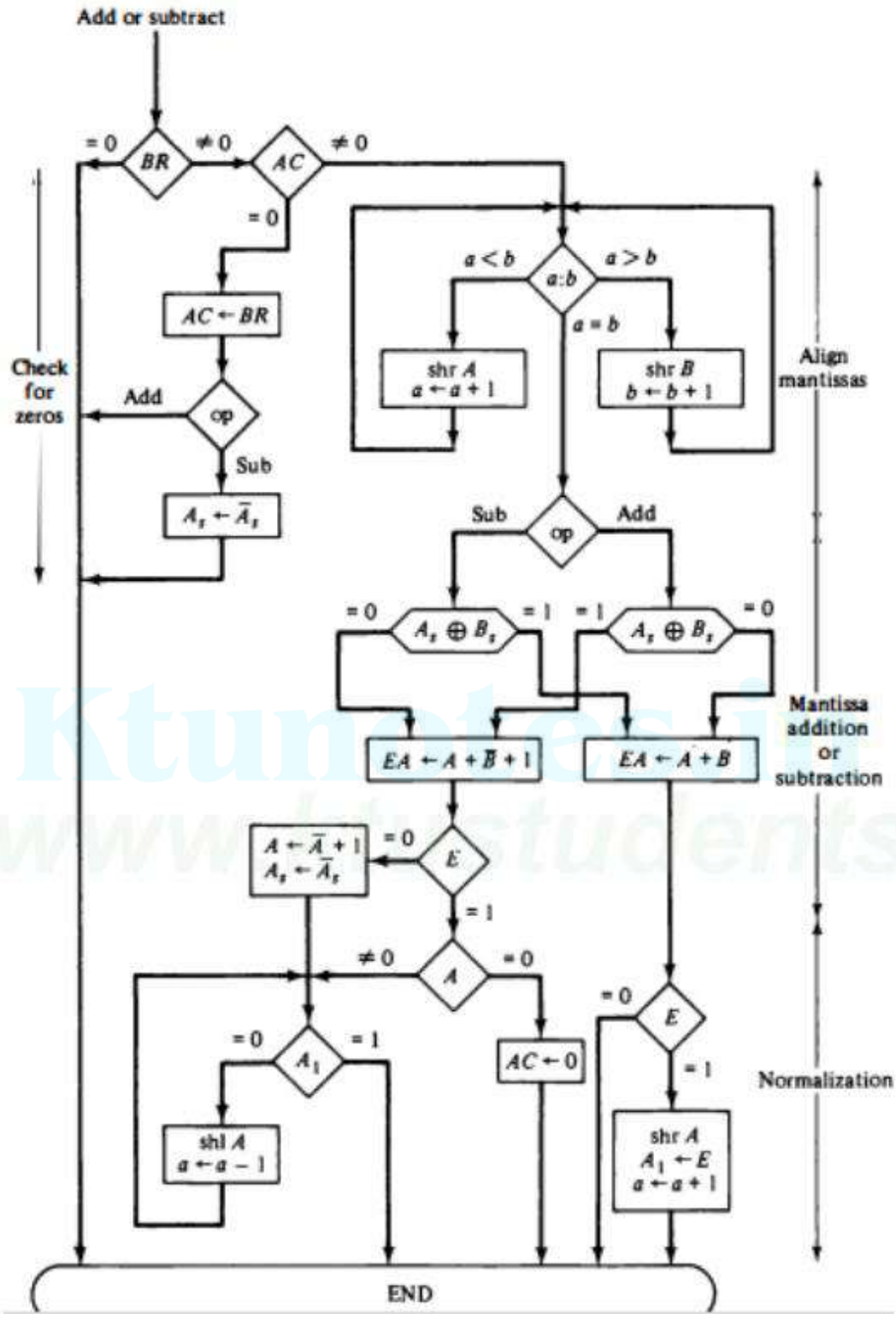
The flowchart for adding or subtracting two floating-point binary numbers is shown in Fig. 10-15. If *BR* is equal to zero, the operation is terminated, with the value in the *AC* being the result. If *AC* is equal to zero, we transfer

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.



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.

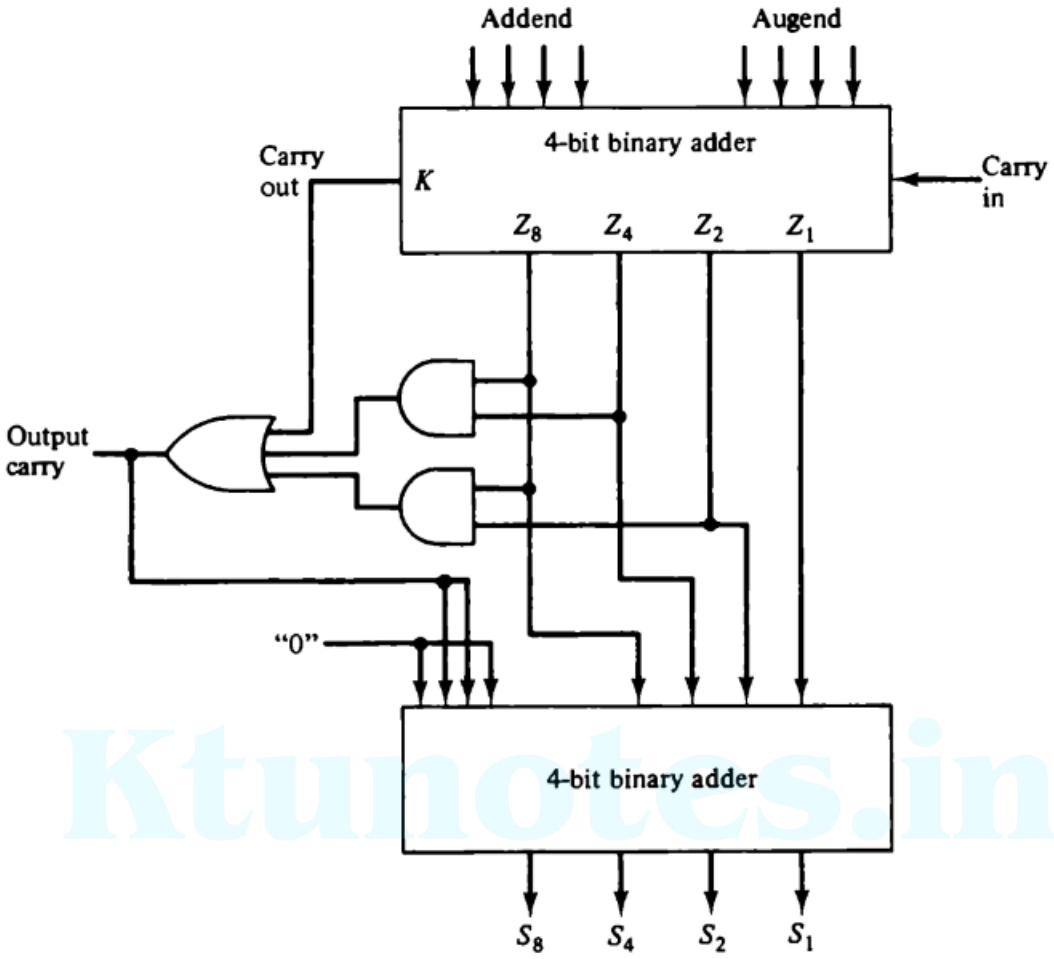
The logic circuit that detects the necessary correction can be derived from the table entries. It is obvious that a correction is needed when the binary sum has an output carry $K = 1$. The other six combinations from 1010 to 1111 that need a correction have a 1 in position Z_8. To distinguish them from 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.



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

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

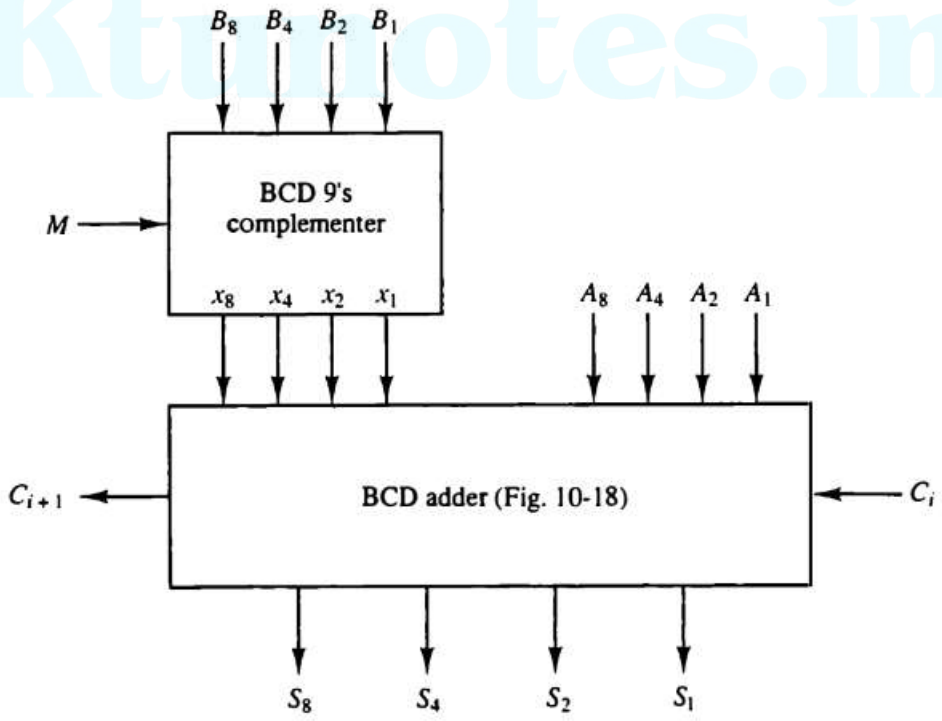
$$x_2 = B_2$$

$$x_4 = B_4 M' + (B_4' B_2 + B_4 B_2')M$$
$$x_8 = B_8 M' + B_8' B_4' B_2' M$$

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.

Figure 10-19 One stage of a decimal arithmetic unit.



MODULE 5

RING COUNTER

Ring Counter is a cascaded connection of flip flops where the ouput of the last flip flop is connected as input to the first flip flop. The output of first flip flop is connected to input of second flip flop and so on.

An n- bit ring counter is implemented using n flip flops and will have n states.

The applications of ring counter are

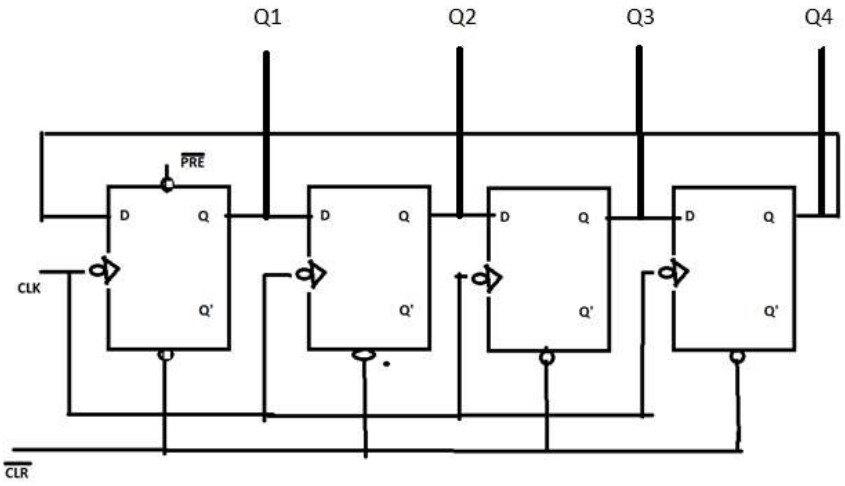
- Frequency counter
- ADC
- Digital clocks
- Measure timers and rate, etc.

- Ring counters are used to count the data in a continuous loop.
- They are also used to detect the various numbers values or various patterns within a set of information, by connecting AND & OR logic gates to the ring counter circuits.
- 2 stage, 3 stage and 4 stage ring counters are used in frequency divider circuits as divide by 2 and divide by 3 and divide by 4 circuits, respectively.

4-bit Ring Counter

4-bit ring counter is implemented using 4 flip flops and has four states-1000, 0100, 0010 and 0001

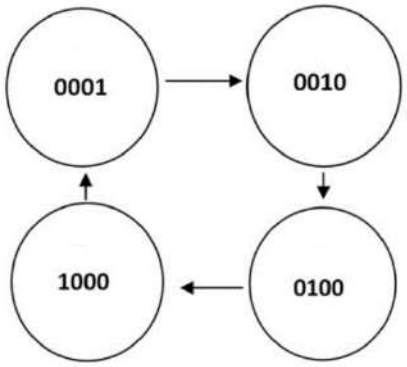
Logic Dircuit Diagram:



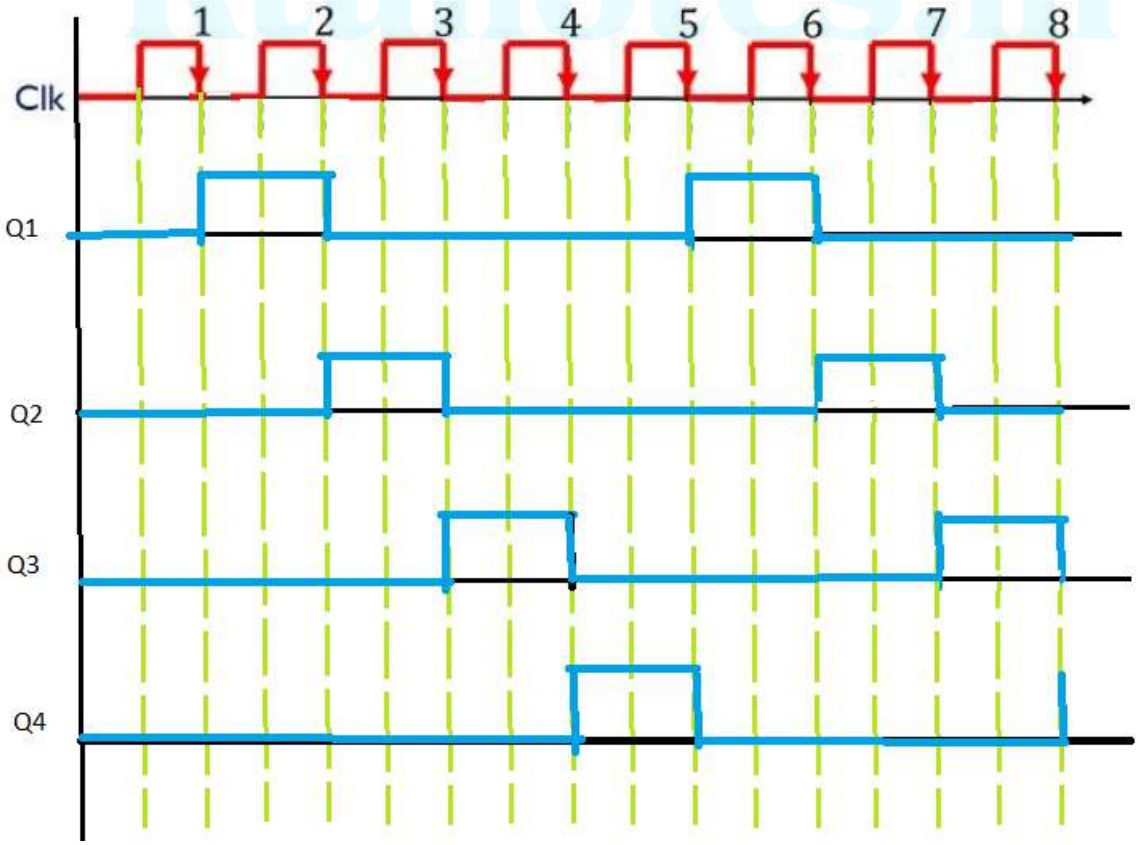
Truth Table:

CLK	Q1	Q2	Q3	Q4
0	0	0	0	0
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

State Diagram:



Timing Diagram:



- **Advantages**
- Can be implemented using D and JK flip-flops.
- It is a self-decoding circuit. It doesn't need a decoder
- **Disadvantages**
- Only four of the 15 states are being utilized.

JOHNSON COUNTER

Johnson Counter is a cascaded connection of flip flops where the inverted ouput of the last flip flop is connected as input to the first flip flop. The output of first flip flop is connected to input of second flip flop and so on.

An n- bit johnson counter is implemented using n flip flops and will have 2n states.

It is also known as an inverse feedback counter or twisted ring counter

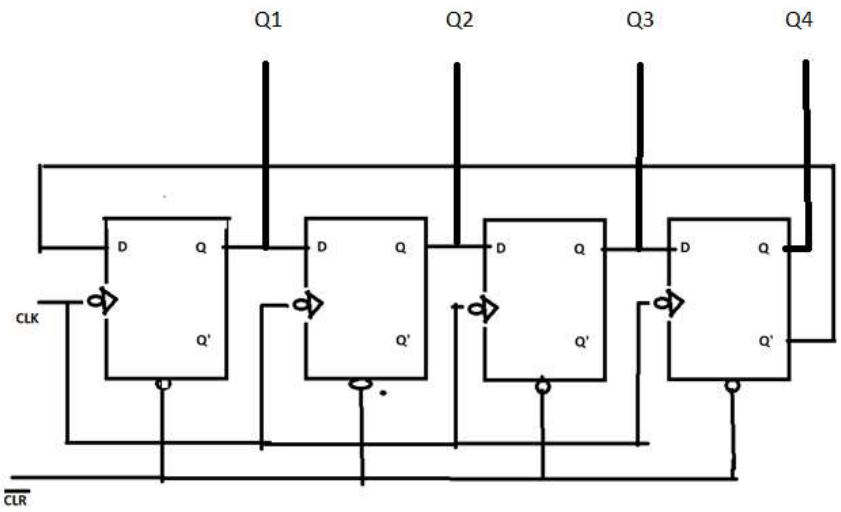
The applications of johnson counter are

- Johnson counters are used as frequency dividers and pattern recognizers.
- It is used as a synchronous decade counter and divider circuit
- It can be used to create complicated finite state machines in hardware logic design.
- The 3-bit johnson counter is used as a 3-phase square wave generator to produce 120 degrees phase shift
- The frequency of the clock signal is divided by varying their feedback.
- The 3 stage Johnson counter is used as a 3 phase square wave generator which produces 1200 phase shift.
- The 5 stage Johnson counter circuit is generally used as synchronous decade (BCD) counter and also as divider circuit.
- The 2 stage Johnson counters are also known as "Quadrature oscillator" which is used to produce 4 level individual outputs which are out of phase with 900 with each other. This quadrature generator is used to produce 4 phase timing signal.

4-bit Johnson Counter

4-bit johnson counter is implemented using 4 flip flops and has 8 states

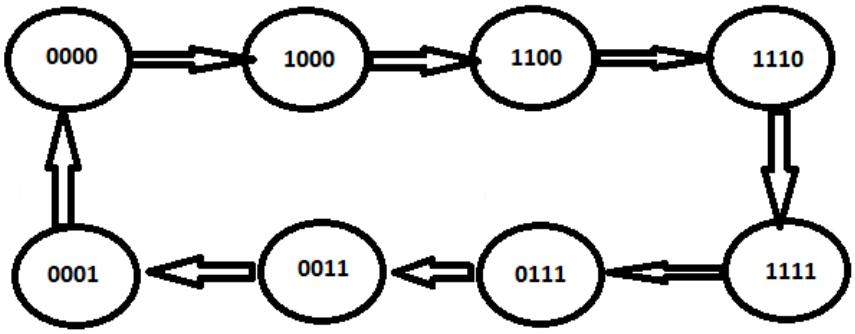
Logic Dircuit Diagram:



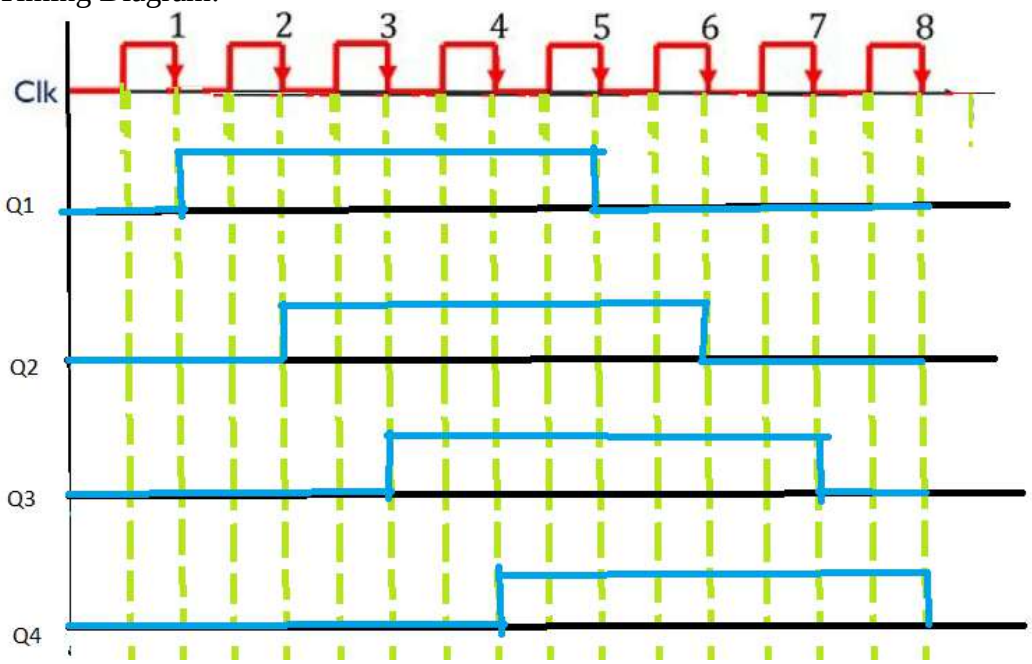
Truth Table:

CLK	Q1	Q2	Q3	Q4
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	1	1	1	1
5	0	1	1	1
6	0	0	1	1
7	0	0	0	1

State Diagram:



Timing Diagram:



- **Advantages**
- More outputs as compared to ring counter.
- It has same number of flip flop but it can count twice the number of states the ring counter can count.
- It count the data in a continuous loop
- It only needs half the number of flip-flops compared to the standard ring counter for the same MOD
- **Disadvantages**
- Only 8 of the 15 states are being used.
- It doesn't count in a binary sequence.

Ring Counter vs Johnson Counter:

BASIS OF COMPARISON	RING COUNTER	JOHNSON COUNTER
Application	Ring counter is mostly used in successive approximation type ADC and stepper motor control.	Johnson counter is also referred to as walking counter or switching tail counter and is mostly used in phase shift or function generator.
Output	In ring counter, the output of the last flip flop is connected to the input of the first flip flop.	In Johnson counter, the output bar or Q-bar of the last flip flop is connected to the input of the first flip flop.
Decoding	Decoding is easy in ring counter as the number of states is equal to the number of flip flops.	Decoding Johnson counter is complex as compared to ring to counter.
Number Of Flip-Flops Vs Number Of Possible States Used	If 'n' is the number of flip flops that is used in ring counter, the number of possible states is 'n'. That means the number of states is equal to the number of flip flops used.	If 'n' is the number of flip flop used in Johnson counter, then the total number of states used is '2n'.

MODULE 5: Shift Register

Flip flops can be used to store a single bit of binary data (1or 0). However, in order to store multiple bits of data, we need multiple flip flops. N flip flops are to be connected in an order to store n bits of data. A Register is a device which is used to store such information. It is a group of flip flops connected in series used to store multiple bits of data.

The information stored within these registers can be transferred with the help of shift registers. Shift Register is a group of flip flops used to store multiple bits of data. The bits stored in such registers can be made to move within the registers and in/out of the registers by applying clock pulses. An n-bit shift register can be formed by connecting n flip-flops where each flip flop stores a single bit of data.

The registers which will shift the bits to left are called “Shift left registers”.

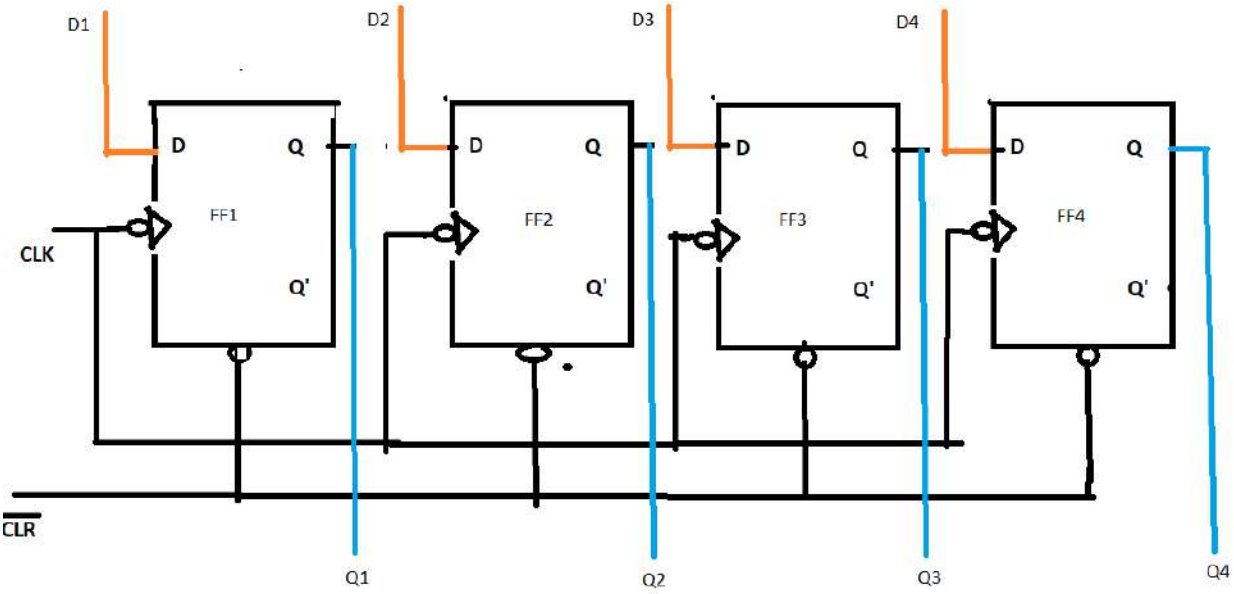
The registers which will shift the bits to right are called “Shift right registers”.

Shift registers are basically of 4 types. These are:

- Serial In Serial Out shift register
- Serial In parallel Out shift register
- Parallel In Serial Out shift register
- Parallel In parallel Out shift register

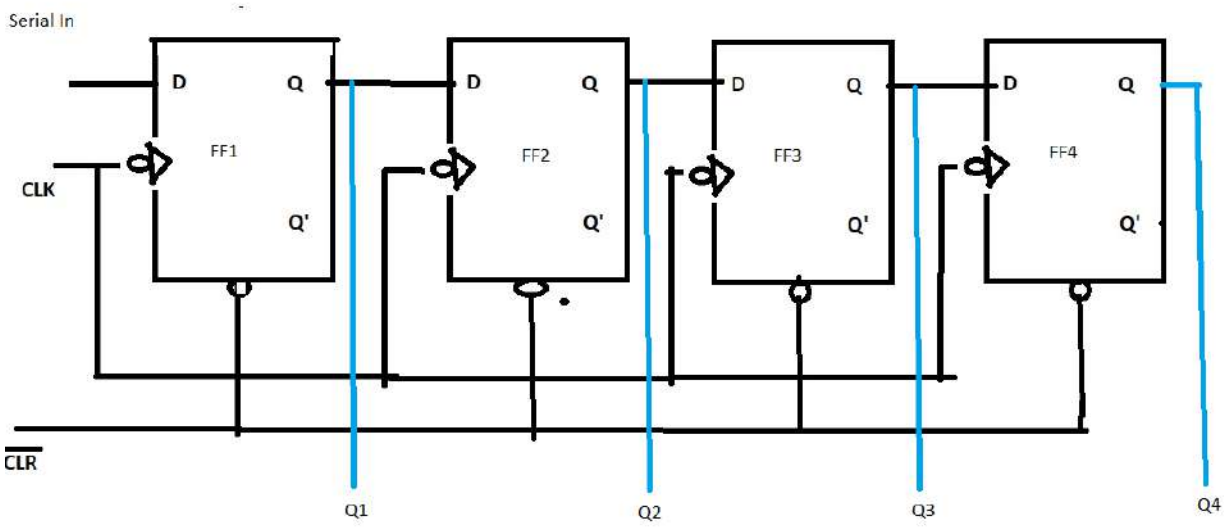
Parallel-In Parallel-Out Shift Register (PIPO) –

The shift register, which allows parallel input (data is given separately to each flip flop and in a simultaneous manner) and also produces a parallel output is known as Parallel-In parallel-Out shift register.



Serial-In Parallel-Out shift Register (SIPO) –

The shift register, which allows serial input (one bit after the other through a single data line) and produces a parallel output is known as Serial-In Parallel-Out shift register.

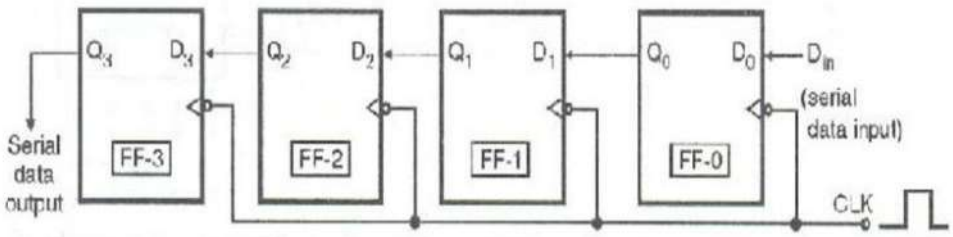


Serial-In Serial-Out Shift Register (SISO) –

The shift register, which allows serial input (one bit after the other through a single data line) and produces a serial output is known as Serial-In Serial-Out shift register. Since there is only one output, the data leaves the shift register one bit at a time in a serial pattern, thus the name Serial-In Serial-Out Shift Register.

The logic circuit given below shows a serial-in serial-out shift register. The circuit consists of four D flip-flops which are connected in a serial manner. All these flip-flops are synchronous with each other since the same clock signal is applied to each flip flop.

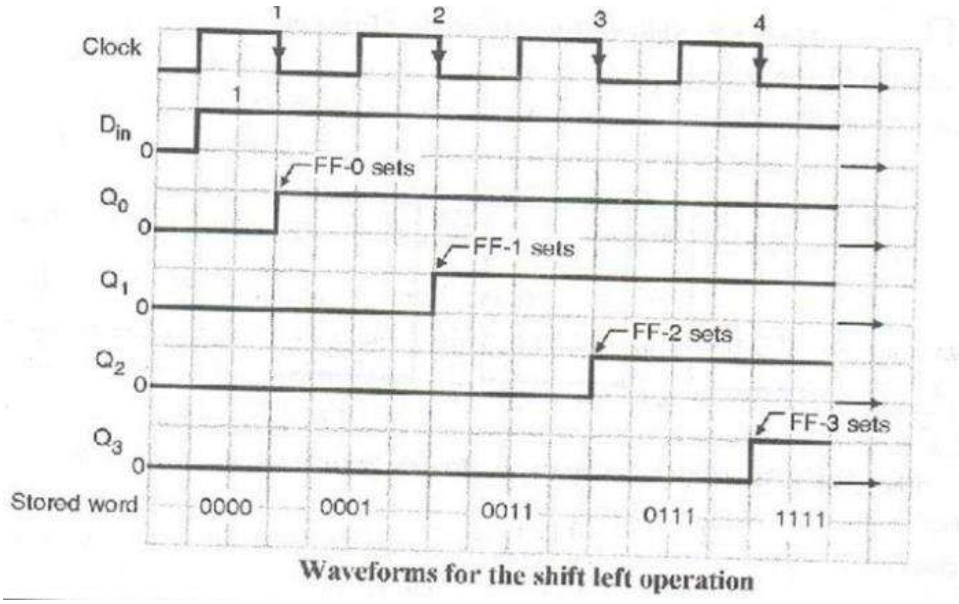
4 Bit Serial Shift left Register



: Serial shift left register

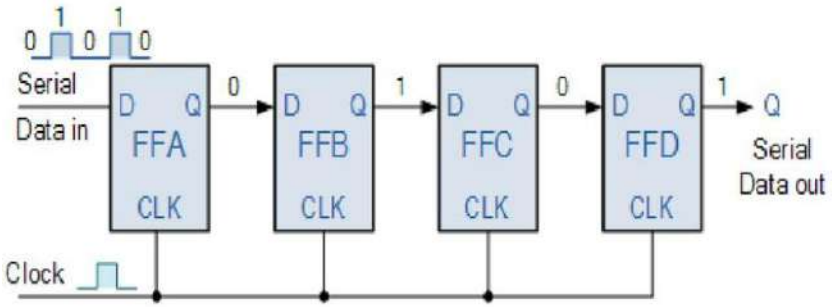
	CLK	Q3	Q2	Q1	Q0	SERIAL INPUT Din = Do
INITIALLY		0	0	0	0	
1st	↓	0	0	0	1	1
2nd	↓	0	0	1	1	1
3rd	↓	0	1	1	1	1
4th	↓	1	1	1	1	1

Direction of data travel ←



Waveforms for the shift left operation

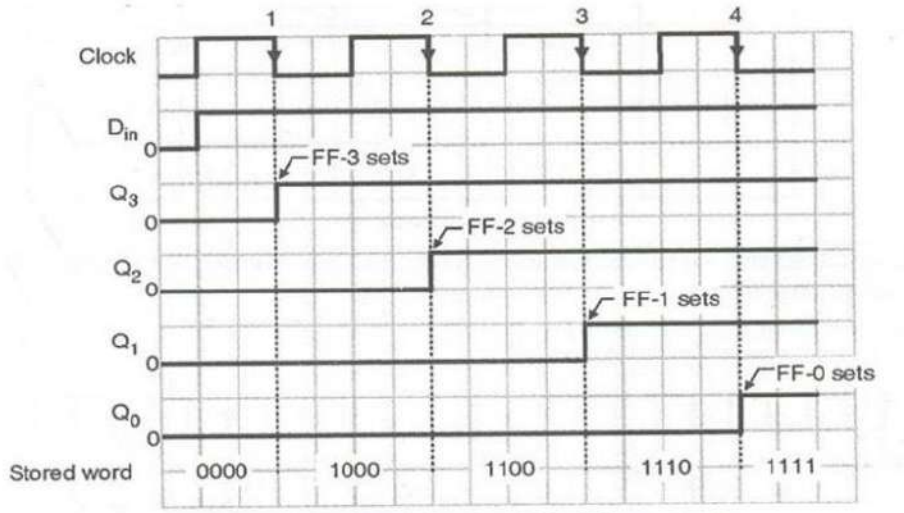
4 Bit Serial Shift Right Register



shift right operation

	CLK	$D_{in} = Q_3$	$Q_3 = D_2$	$Q_2 = D_1$	$Q_1 = D_0$	Q_0
Initially			0	0	0	0
1st	↓	1	1	0	0	0
2nd	↓	1	1	1	0	0
3rd	↓	1	1	1	1	0
4th	↓	1	1	1	1	1

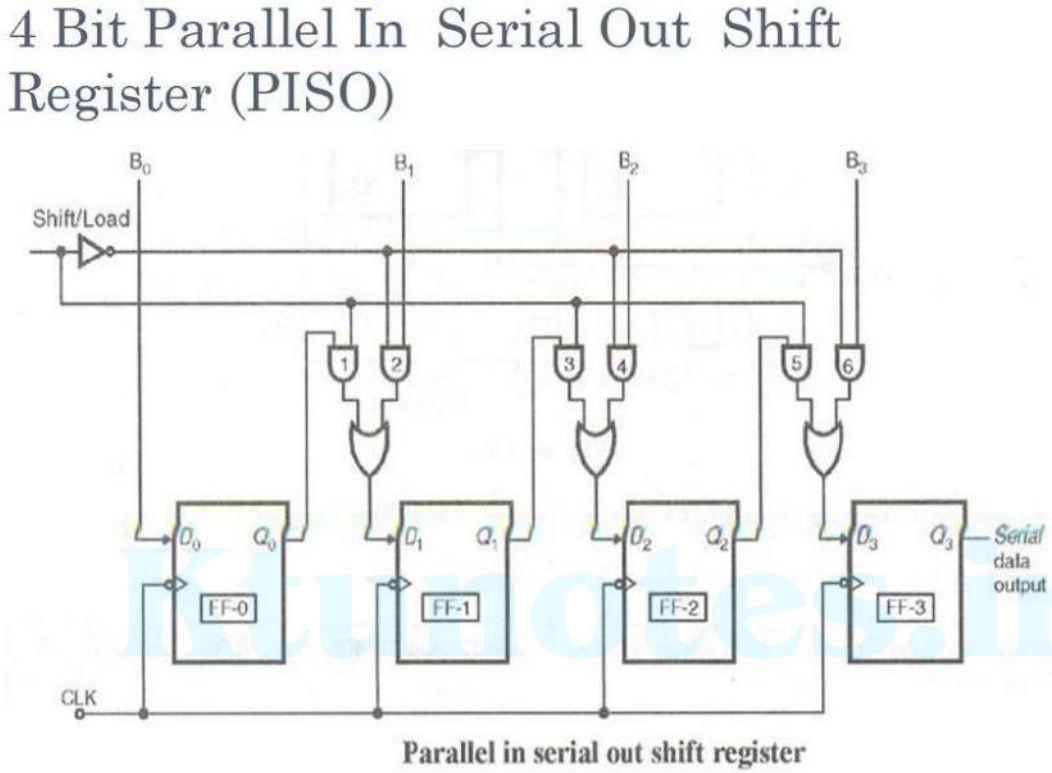
Direction of data travel



Waveforms for the shift right operation

Parallel-In Serial-Out (PISO)-

The shift register, which allows parallel input (data is given separately to each flip flop and in a simultaneous manner) and produces a serial output is known as Parallel-In Serial-Out shift register.



Parallel in serial out shift register

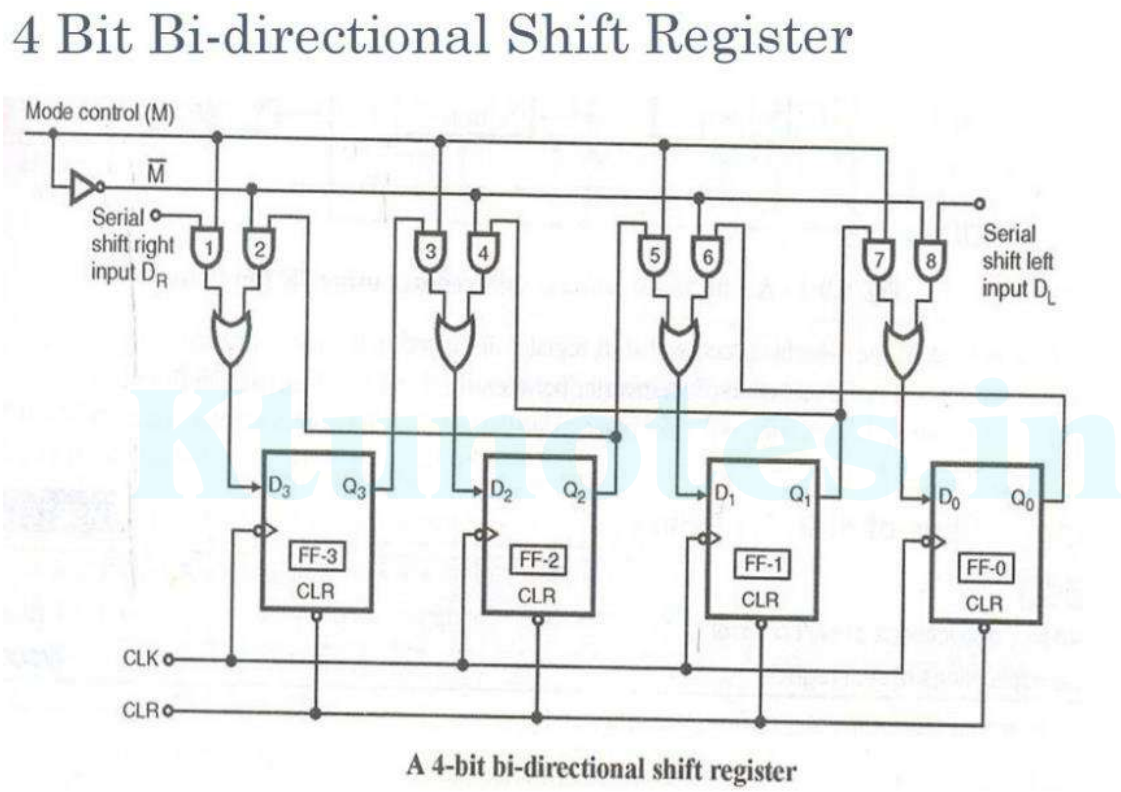
- Load mode:
- Load mode s active low device, they gives s/g to AND gate 2,4,6 & they become active. They pass B0,B1,B2,B3 bits to the corresponding flipflops.
- On the low going edge of clock the binary inputs B0,B1,B2,B3 will get loaded into the corresponding flipflops. Thus parallel loading is take place.

- Shift mode:
- Shift mode is active high device, they gives s/g to AND gate 1,3,5 & they become active at that time AND gate 2,4,6 become inactive. Hence parallel loading is not possible therefore data shifting of data from left to right, bit by bit.

Bidirectional Shift Register –

If we shift a binary number to the left by one position, it is equivalent to multiplying the number by 2 and if we shift a binary number to the right by one position, it is equivalent to dividing the number by 2.To perform these operations we need a register which can shift the data in either direction.

Bidirectional shift registers are the registers which are capable of shifting the data either right or left depending on the mode selected. If the mode selected is 1(high), the data will be shifted towards the right direction and if the mode selected is 0(low), the data will be shifted towards the left direction.

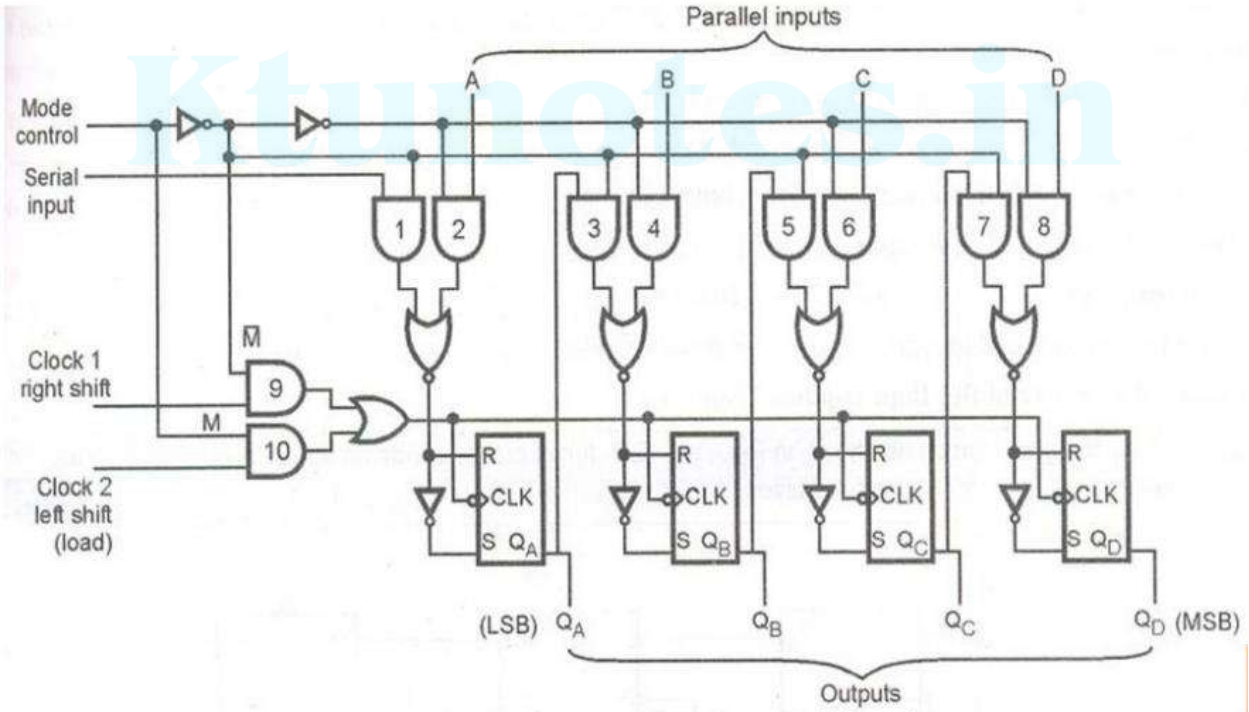


A 4-bit bi-directional shift register

- M=1= Shift right operation
- M=1 then AND gate 1,3,5,7 are enabled whereas the remaining AND gate 2,4,6,8 will be disable.
- Data D_R shifted bit by bit from FF3 to FF0, on the application of clock pulse.

- M=0=Shift left operation
- M=0 then AND gate 2,4,6,8 are enabled whereas the remaining AND gate 1,3,5,7 will be disable.
- Data D_L shifted bit by bit from FF0 to FF3, on the application of clock pulse.
- M changed only when clock =0 otherwise data stored in the register may be altered.

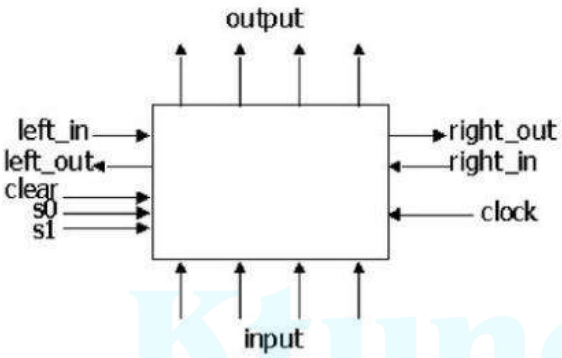
Universal Shift Register



Logic diagram of a universal shift register

- This shift register is capable of performing the following operations
- Parallel loading
- Left shifting
- Right shifting

- Holds 4 values
  - serial or parallel inputs
  - serial or parallel outputs
  - permits shift left or right
  - shift in new values from left or right



clear sets the register contents and output to 0

s1 and s0 determine the shift function

s0	s1	function
0	0	hold state
0	1	shift right
1	0	shift left
1	1	load new input