(2) Topics.

UNIT-IV

COMPUTER ARITHMETIC

Addition and Subtraction :-

We can perform addition and subtraction of two binary numbers in three different ways:

1) Signed - Magnitude representation
2) Signed - 1's Complement "
3) Signed - 2's "

The Signed - 2's complement representation is used by performing arithmetic operations. The signed - magnitude representation is used for floating - point representation.

Addition and Subtraction with Signed - Magnitude data :-

Here, we are going to consider the magnitude of any two numbers i.e., A and B. There are eight different operations which are listed in below:

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)

The 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.

Algorithm :-

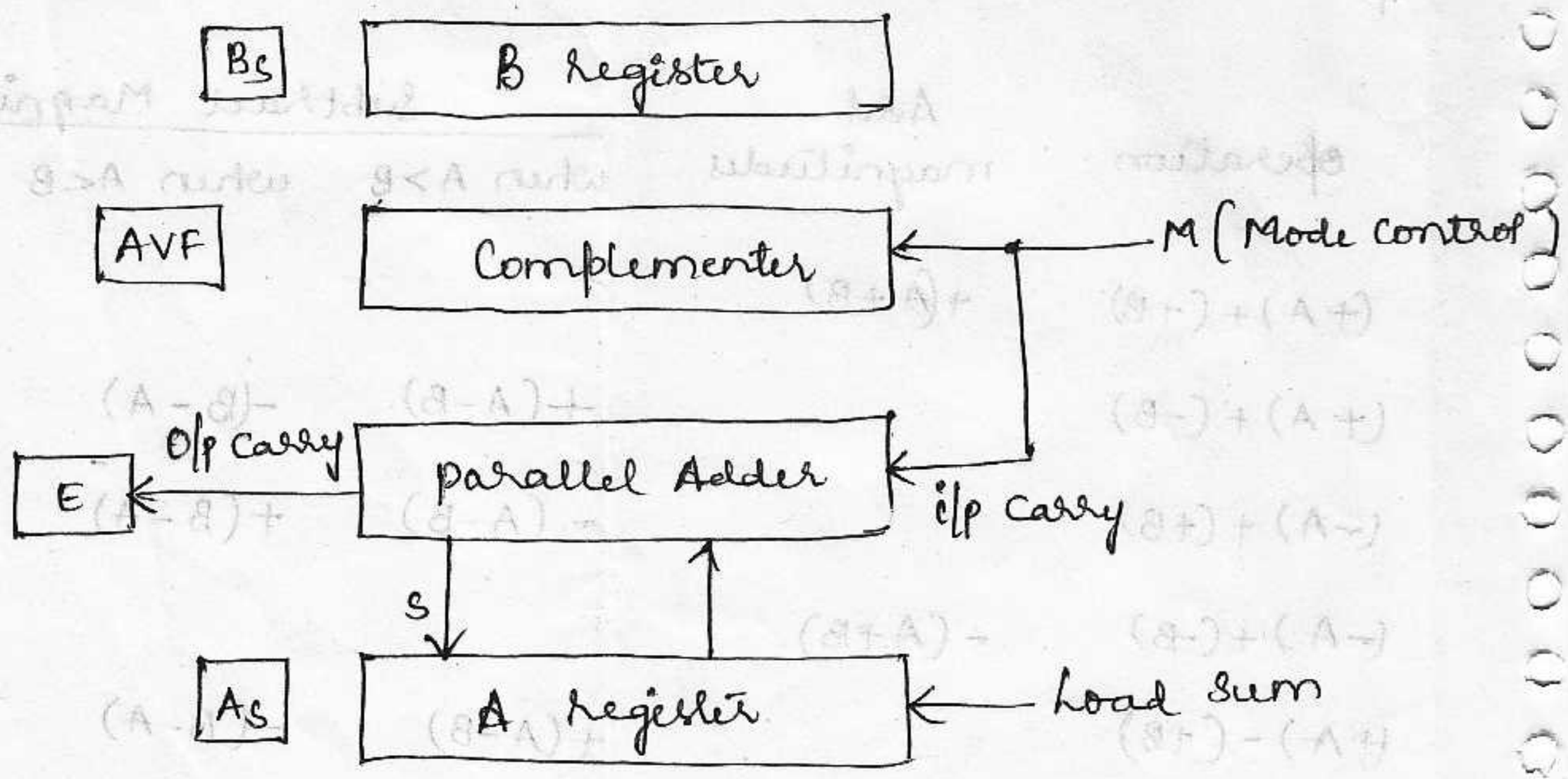
When the signs of A and B are identical, add the two magnitudes and attach the sign of A to the result.

When the signs of A and B are different, Compare the magnitudes, Subtract the smaller number from the larger.

Hardware implementation :-

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

The h/w implementation for addition & Subtraction is shown in fig.



Let A and B be two registers that holds the magnitudes of the numbers and As & Bs be two flip-flops that hold the corresponding signs. The result of the

operation may be transferred to a third register. or the result is transferred into A or A_S.

First, a parallel-adder is needed to perform micro operation A+B. Second a Comparator circuit is needed to establish if A>B, A=B or A<B. The third, two parallel-subtractor circuits are needed to perform the micro operations, A-B & B-A.

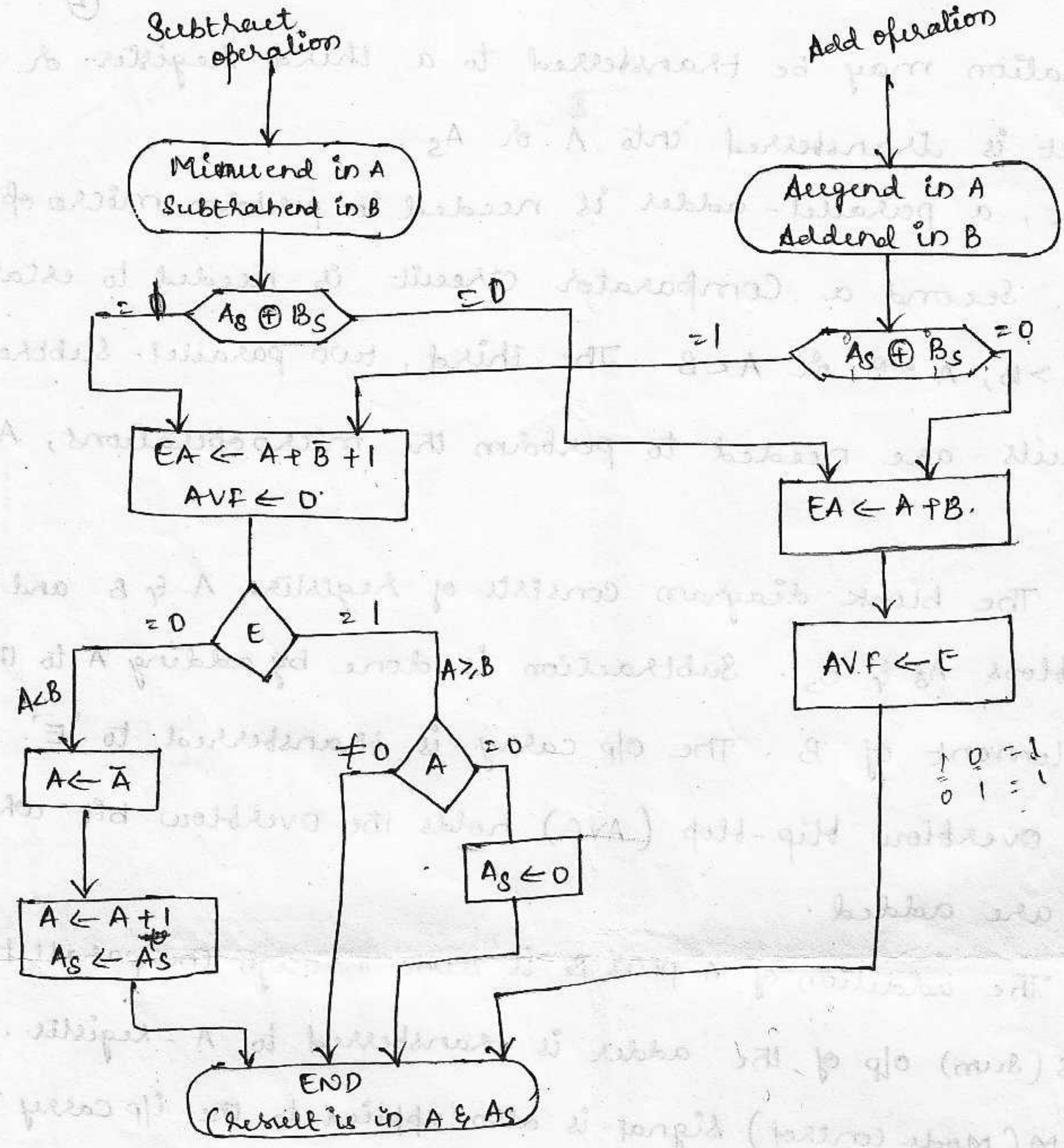
The block diagram consists of registers A & B and sign flip-flops A_S & B_S. Subtraction is done by adding A to the 2's Complement of B. The o/p carry is transferred to 'E'. The add overflow flip-flop (AVF) holds the overflow bit when A & B are added.

The addition of A plus B is done through the parallel Adder. The S (Sum) o/p of the adder is transferred to A-register. The M (Mode control) signal is also applied to the i/p carry of the adder. When M=0, the o/p of B is transferred to the adder, the i/p carry is '0' and the o/p of the adder is equal to the sum A+B. When M=1, the 1's complement of B is applied to the adder, the i/p carry is 1, o/p $S = A + \bar{B} + 1$

Hardware Algorithm:-

The flowchart for the h/w algorithm is shown in fig.

The two signs A_S & B_S are compared by EX-OR gate. If the o/p of the gate is 0, the signs are identical. If the o/p of the gate is 1, the signs are different. For an add operation, the



Identical signs dictate that the magnitudes be added. For a subtract operation, different signs dictate that the magnitudes be added. The magnitudes are added ~~where~~ with EA ← A+B, where EA is a register that combines E & A.

The two magnitudes are subtracted if the signs are different for an add operation or identical for a subtract operation. No overflow can occur if the numbers are subtracted so AVF is to '0'.

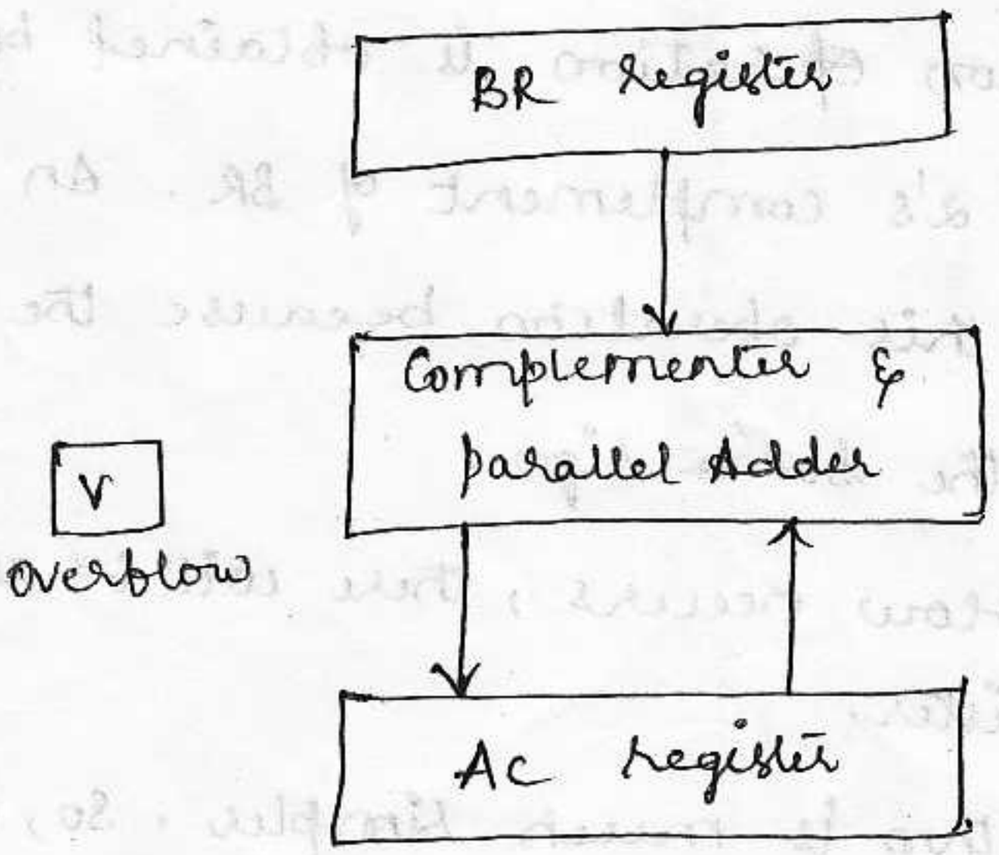
If E=1, then the condition is A ≥ B, if A is '0' then the result is correct result. If E=0, then the condition is

Condition is A<B. For this, we are going to consider the 2's complement of the value in A. This operation can be done one micro operation $A \leftarrow \bar{A}+1$. If the sign of the result is as same as the sign of A, so no change in As is required. When A<B, the sign of the result is the complement of the original sign of A. The complement of As is required to get the correct sign. The final result is found in register A & its sign in As.

Addition & Subtraction with signed 2's complement data :-

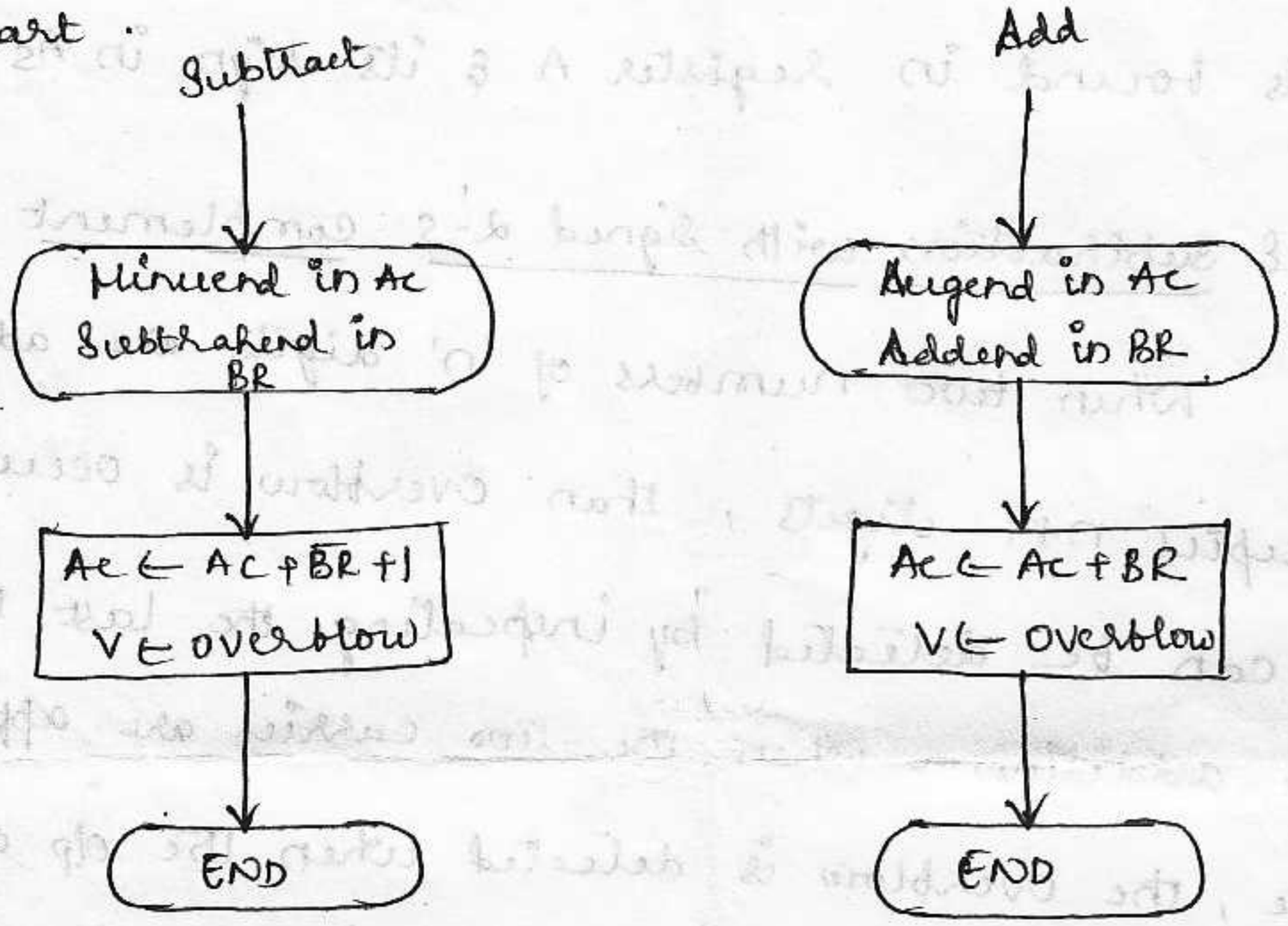
When two numbers of 'n' digits are added and the sum occupies n+1 digits, than overflow is occured. An overflow can be detected by inspecting the last two carries out of the addition. when the two carries are applied to an ex-or gate, the overblow is detected when the o/p of the gate is equal to 1.

The reg. configuration for the h/w implementation is shown in fig.



The leftmost bit in Ac & BR represent the sign bits of the number. The two sign bits are added or subtracted together with the other bits in the complementer & parallel Adder. The overflow flip-flop V is set to 1 if there is an overflow.

The algorithm for adding and subtracting two binary numbers is signed - 2's complement representation is shown in flow chart.



The sum is obtained by adding the contents of AC & BR. The overflow bit is set to 1 if the ex-or of the last two carries is set to 1.

The subtraction operation is obtained by adding the Content of Ac to the 2's complement of BR. An overflow must be checked during this operation because the two numbers added could have the same sign.

If an overflow occurs, there will be an erroneous result in the Ac register.

This algorithm is much simpler. So, most computers uses the signed-magnitude representation.

Multiplication Algorithms :-

The sign of the product is determined from the signs of the multiplicand and multiplier. If they are like, the sign of the product is +ve, If they are unlike, the sign of the product is -ve.

H/w implementation for Signed-Magnitude Data :-

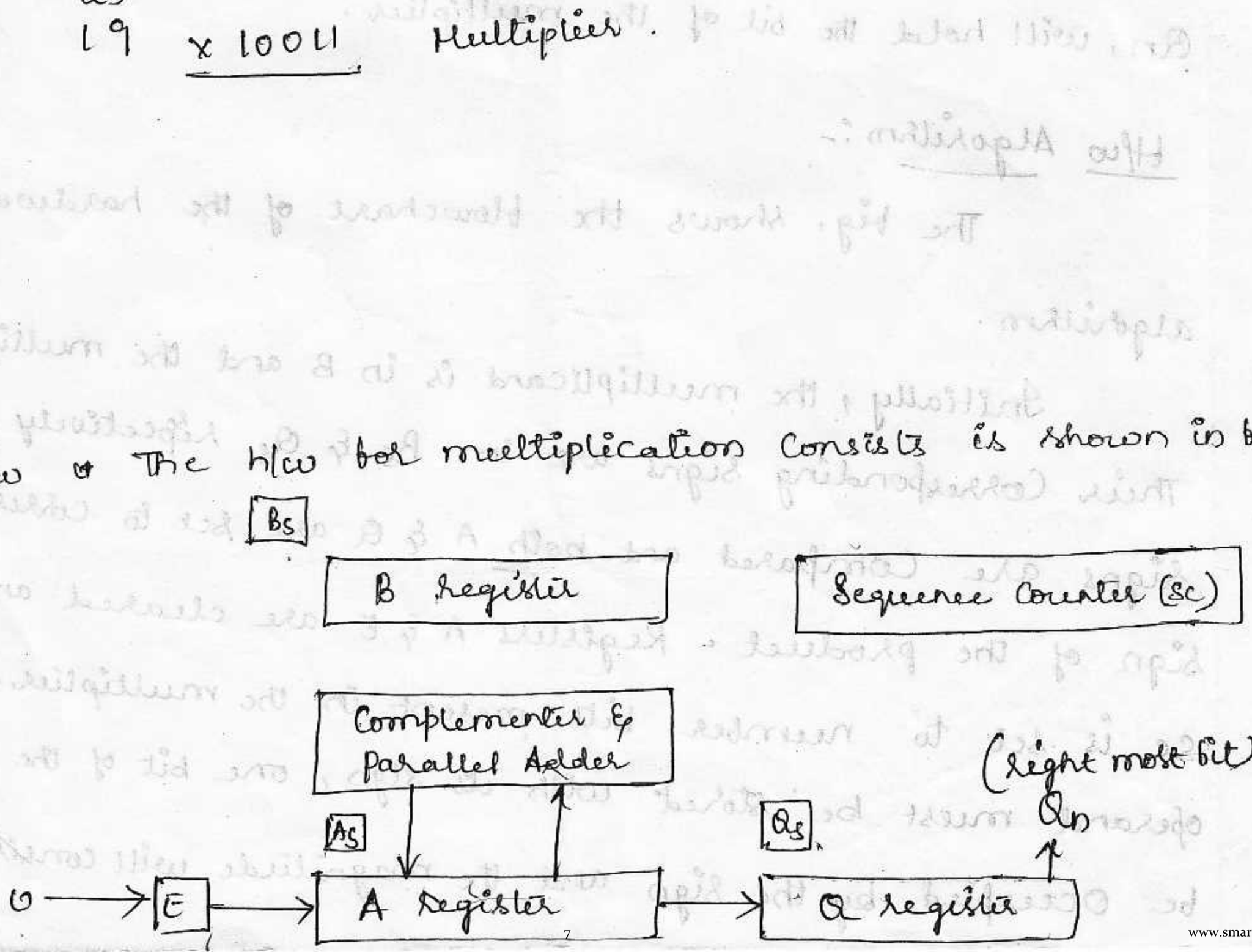
The Multiplication is implemented in a digital computer First, instead of providing registers to store & add, it is convenient to provide an adder for the summation of only two binary numbers and accumulate the partial products in a register.

Second, instead of shifting the multiplicand to the left, the partial product shifted to right.

Third, when the corresponding bit of the multiplier is 0, there is no need to add all zeros to the partial product.

Eg: 23 10111 Multiplicand
19 x 10011 Multiplier.

H/w :- The h/w for multiplication consists is shown in fig.



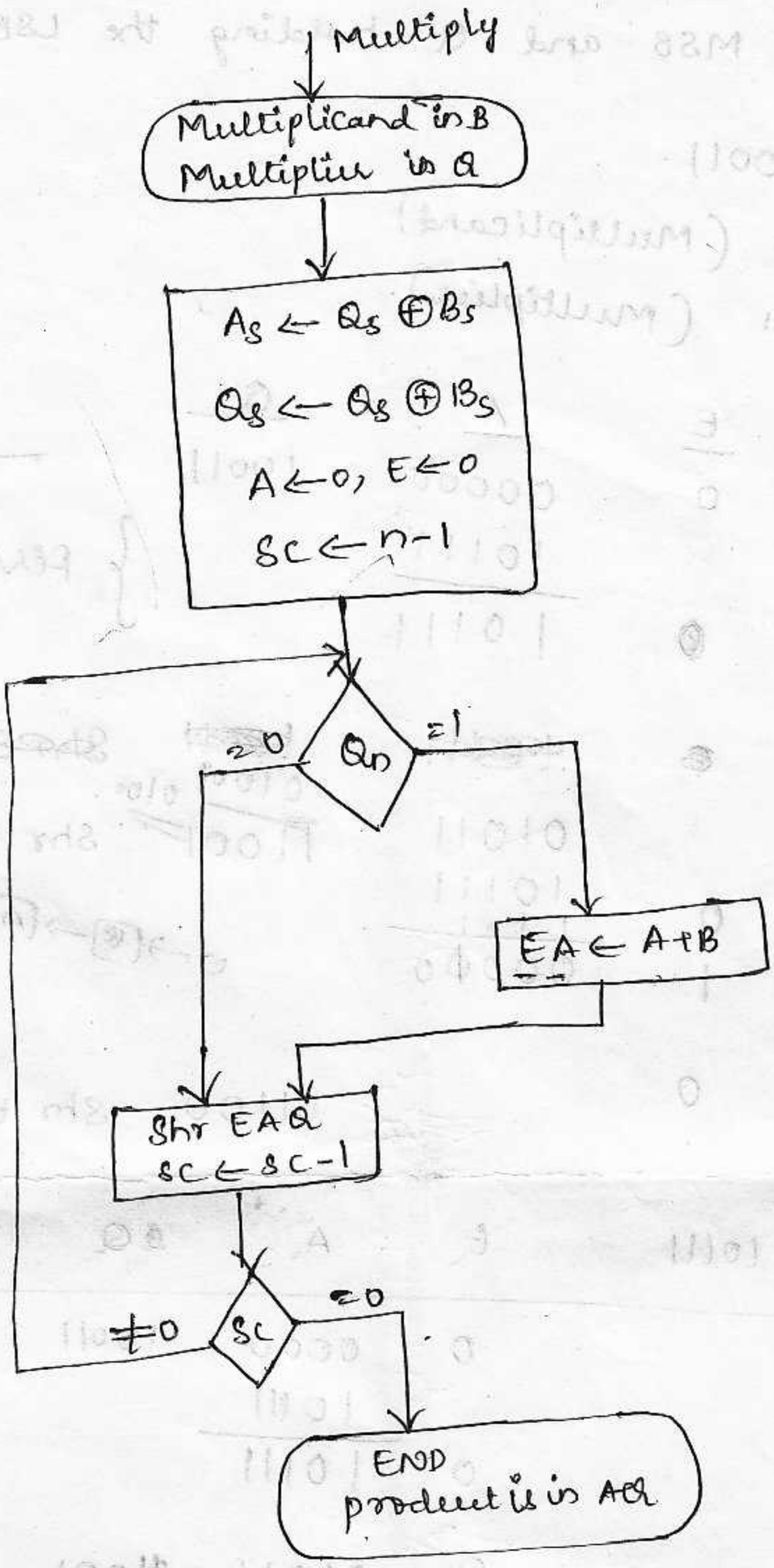
The multiplier stored in the Q register & its sign in Qs The Sequence Counter (SC) is initially set to a number equal the number of bits in the multiplier. The Counter is decremented by 1 after forming each partial product. The Counter reaches to zero, the product is formed and the process stops.

Initially, the multiplicand is in B reg. and the multiplier is Q. The sum of A & B forms a partial product which is transferred to the EA register. Both partial product & multiplier are shifted to the right. This shift will be denoted by the statement Shr EAQ. The LSB of A is shifted into MSB of Q the bit from E is shifted into the MSB of A and '0' is shifted into E. After the shift, one bit of the partial product is shifted into Q, pushing the multiplier bits one position to the right. The right most flip-flop in register Q, designated by Qn, will hold the bit of the multiplier.

H/w Algorithm :-

The fig. shows the flowchart of the hardware multiply algorithm.

Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in Bs & Qs respectively. The signs are compared and both A & Q are set to correspond to the sign of the product. Registers A & E are cleared and Sequence Sc is set to number bits present in the multiplier. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consists of n-1 bits.



After initialization, the low-order bit of the multiplier Qn is tested. If it is 1, the multiplicand in B is added to the present partial product in A. If it is 0, nothing is done, Register EAQ is then shifted once to the right to form the new partial product. The sequence counter is decremented by 1 and its new value is checked. If it is not equal to zero, the process is repeated and if is equal to zero, the process stops. The final product is available in both A & Q, with

A holding the MSB and Q holding the LSB.

Ex: 10111 x 10011.

B = 10111 (Multiplicand)

Q = 10011 (Multiplier).

B	Sc	E	A	Q	
10111	5	0	00000	10011	— Initial values
			10111		
		0	10111		} First partial product

4 0 01011 01001 Shr EAQ
1 10111
1 1 1 1
0 0 0 0 0

0 → E → A → Q } Second partial product

0 01100 Shr EAQ] x

Multiplicand B = 10111	E	A	BQ	Sc	
	0	0000	10011	101	} First partial product
Qn=1 Add		10111			
	0	10111			
Shr EAQ	0	01011	01001	100	} Second partial product
Qn=1, Add		10111			
	1	00010			
Shr EAQ	0	10001	01100	011	
Qn=0, Shr EAQ	0	01000	10110	010	
Qn=0, Shr EAQ	0	00100	01011	001	
Qn=1, Add B		10111			
	0	11011	01011	001	
& Shr, EAQ		01101	10101	000	

AQ = 01101 10101

Booth Multiplication Algorithm :-

Booth algorithm gives a procedure for multiplying binary integers in signed - 2's complement representation. It operates on the fact that strings of 0's in the multiplier requires no addition but just shifting and a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} - 2^m$.

Eg: The binary number 001110 has a string of 1's from 2^3 to 2^1. Here $k=3$, $m=1$. The number can be represented as $2^{k+1} - 2^m$

$= 2^4 - 2^1 = 14$. Therefore, the multiplication $M \times 14$, where M is the multiplicand and 14 is the multiplier. This can be done as $M \times 2^4 - M \times 2^1$, thus the product can be obtained by shifting the binary Multiplicand M four times to the left and subtracting 'M' shifted left once.

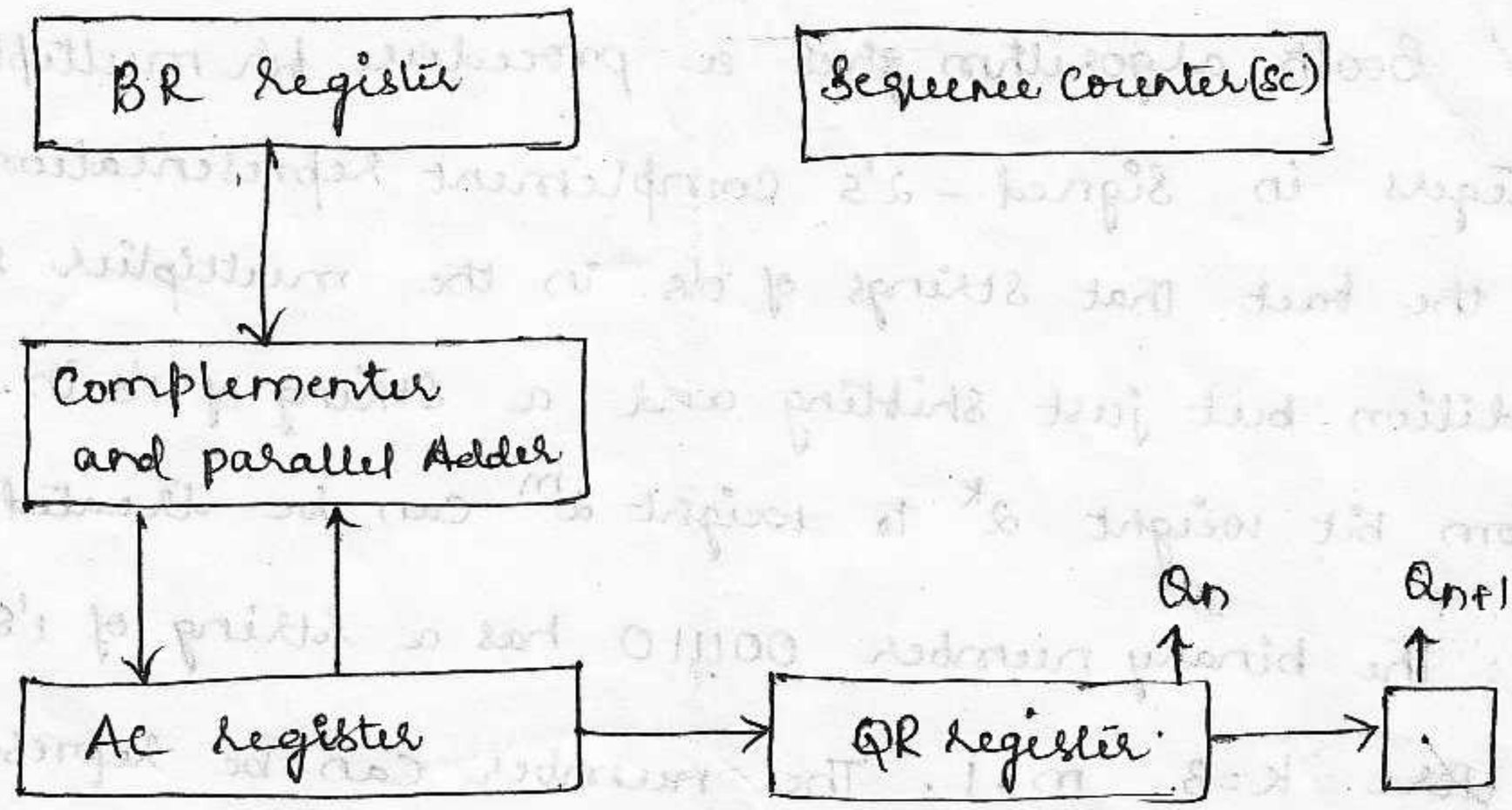
Booth algorithm requires examination of the multiplier bits and shifting the partial product. The following rules are to be required.

1) The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier.

2) The multiplicand is added to the partial product upon encountering the first 0 in a string of 0's in the multiplier.

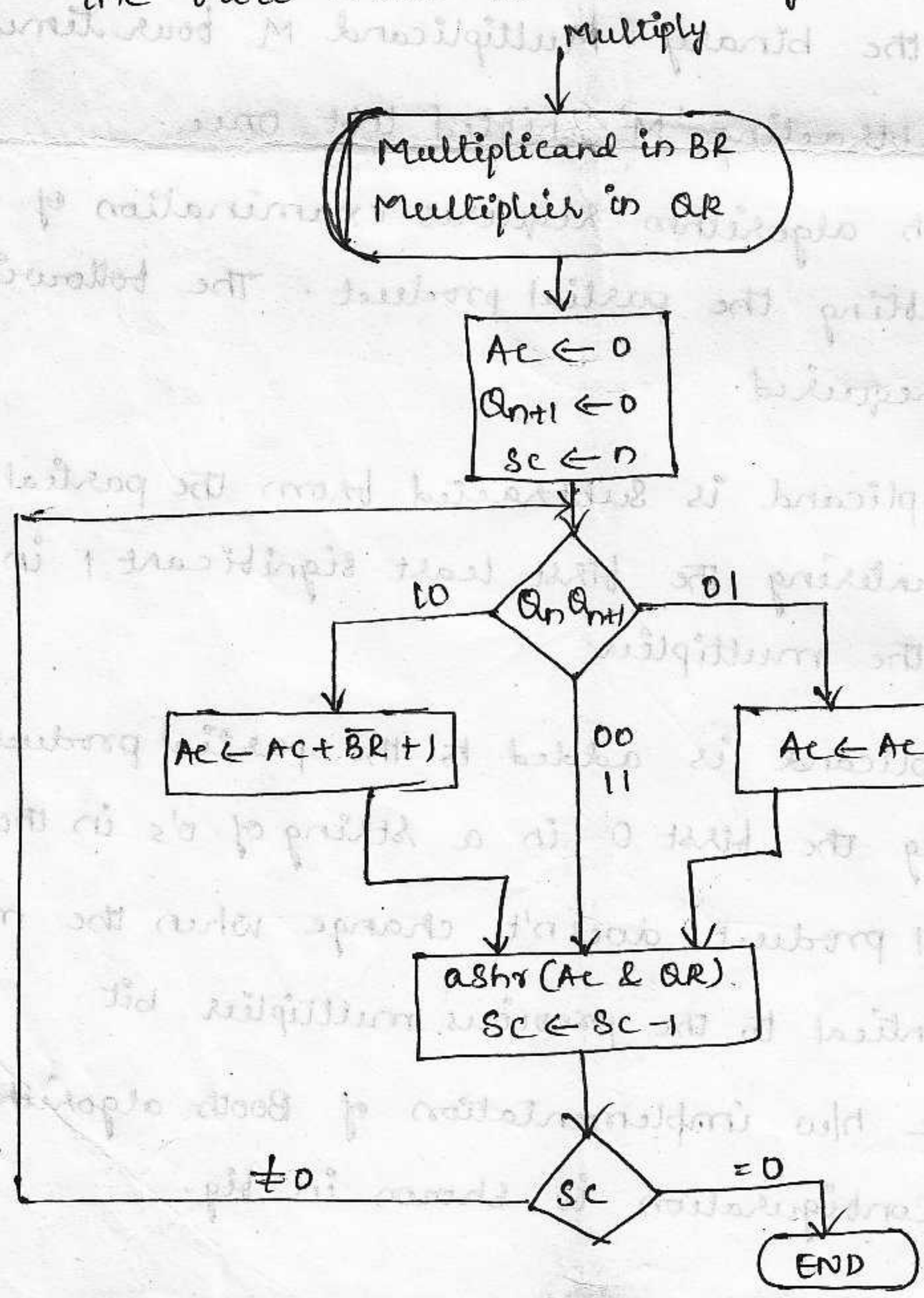
3) The partial product doesn't change when the multiplier bit is identical to the previous multiplier bit.

The h/w implementation of Booth algorithm requires the register configuration is shown in fig.



Here an extra flip-flop Q_{n+1} is appended to QR to facilitate a double bit inspection of the multiplier.

The flow chart for Booth algorithm is shown in fig.



Ex: $(-9) \times (-13) = +117$.

$BR = 10111$

$\overline{BR}+1 = 01001$

Q_n	Q_{n+1}		AC	QR	Q_{n+1}	SC
1	0	Initial	00000	10011	0	101
		Subtract	01001			
			01001			
		ashr	00100	11001	1	100
1	1	ashr	00010	01100	1	011
0	1	Add BR	10111			
			11001			
		ashr	11100	10110	0	010
0	0	ashr	01110	01011	0	001
1	0	Subtract BR	01001			
			10111			
		ashr	01011	10101	1	000

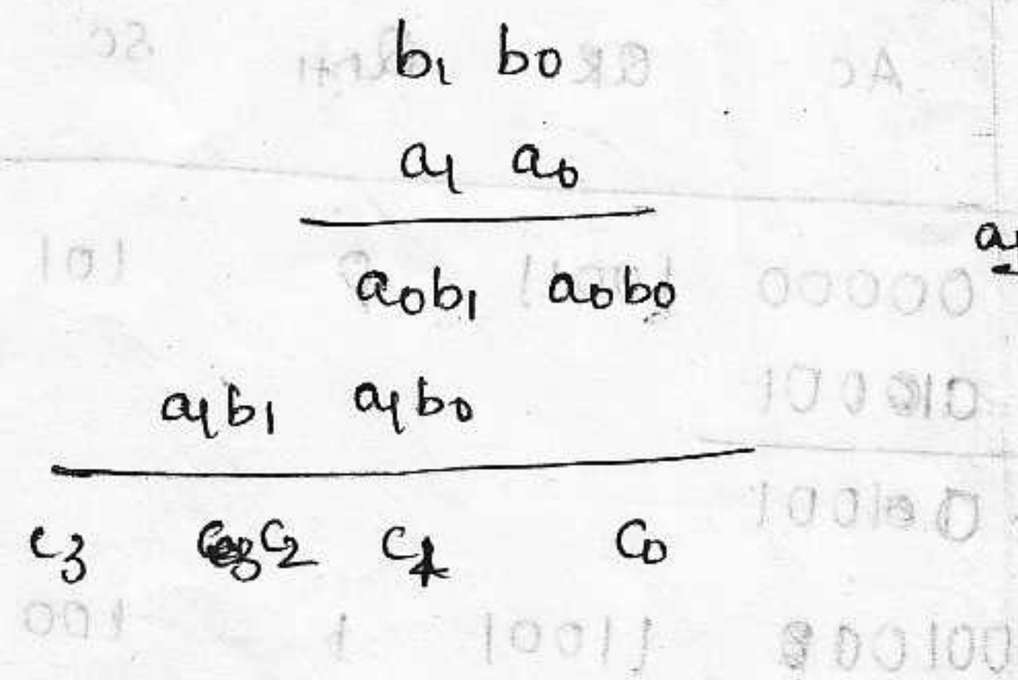
AC QR = 0101110101

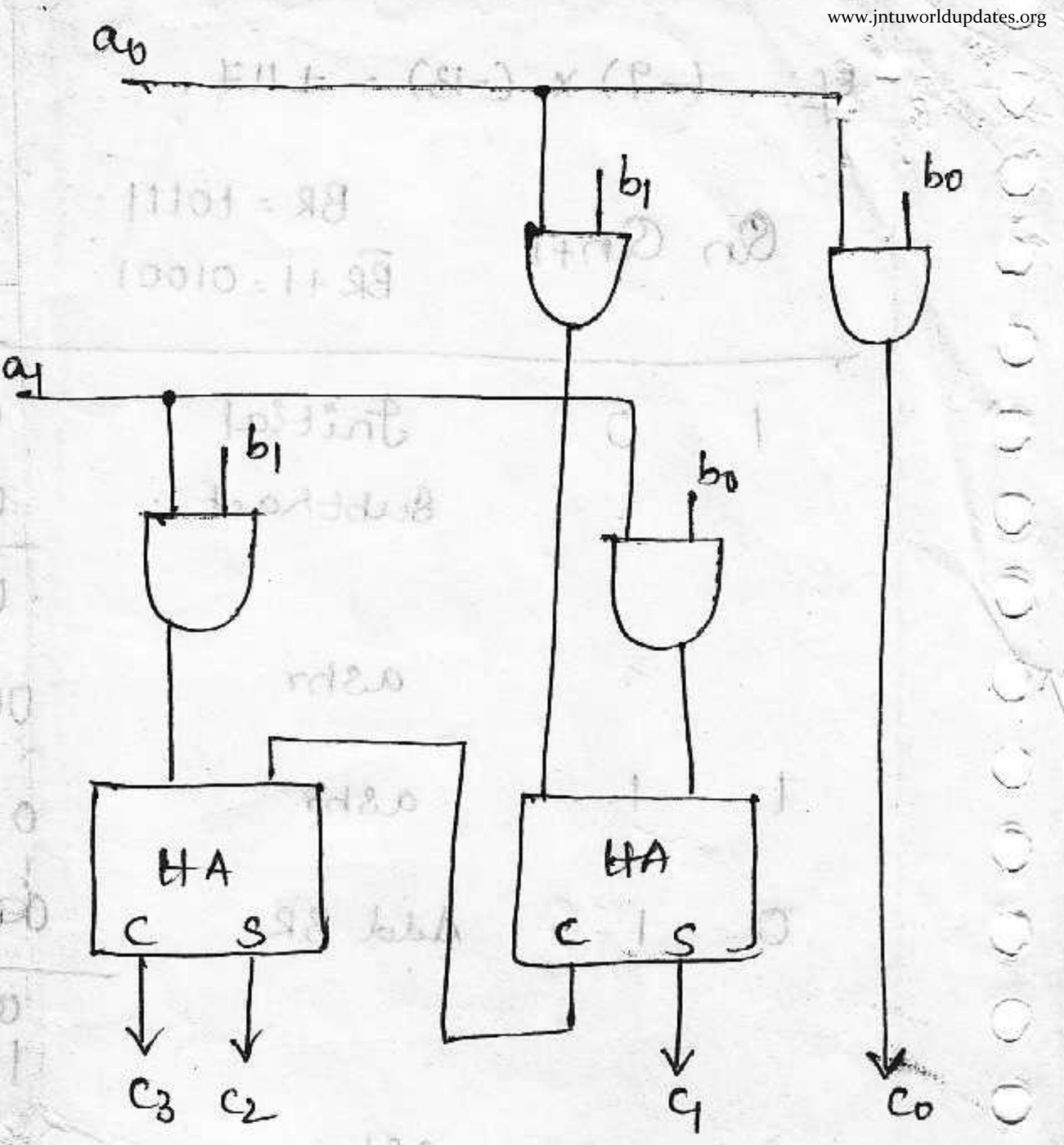
$2^9 \quad 2^5 \quad 2^4 \quad 2^2 \quad 2^0$

Array Multiplier :-

Checking the bits of the multiplier one at a time and forming partial products is a sequential operation that requires a sequence of add and shift micro operations. The multiplication of two binary numbers can be done with one micro operation by means of a combinational circuits that forms the product bit all at once. This is a best way of multiplying two numbers.

An array multiplier can be implemented with a combinational circuit as shown in fig.





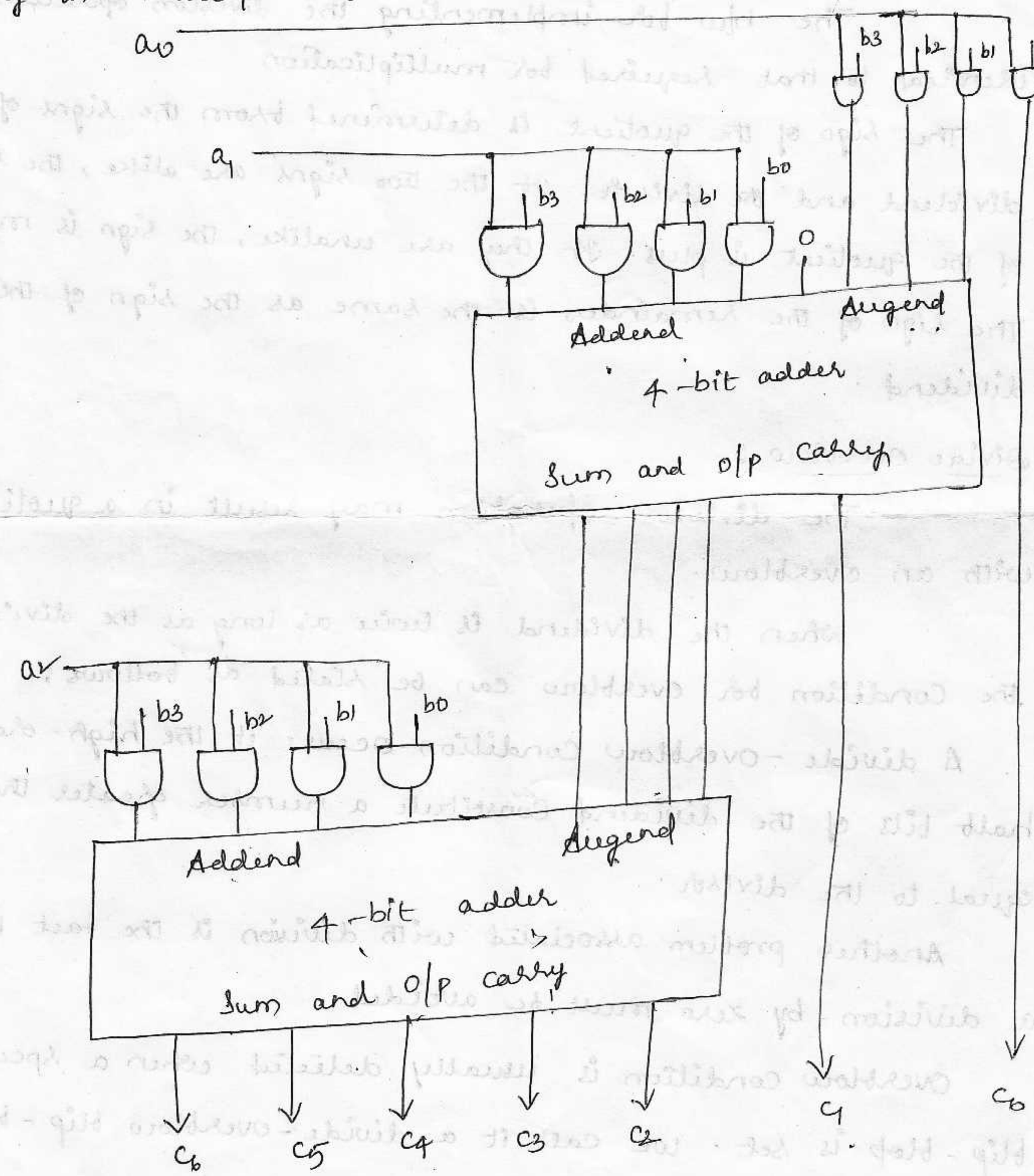
The multiplicand bits are b_1 & b_0, the multiplier bits a_1 & a_0 and the product is C_3 C_2 C_1 C_0. The first partial product is formed by multiplying a_0 by b_1 b_0. The multiplication of two bits a_0 & b_0 produces '1' if both bits are 1, otherwise, it produces 0. This is identical to 'AND' operation.

The second partial product is formed by multiplying a_1 by b_1 b_0 and is shifted one position to the left. The two partial products are added with two half adders (HA) circuits. It is necessary of half adders because to produce sum.

A combinational circuit binary multiplier with more bits can be constructed in a similar fashion. The binary output in each level of AND gates is added parallel with the partial product of the previous level to form a new partial product

For j multiplier bits & 'k' multiplicand bits we need j x k AND gates and (j-1) k-bit adders to produce a product of j+k bits.

As a second example, consider a multiplier circuit that multiplies a binary number of four bits with a number of three bits. The multiplicand is represented by $b_3 b_2 b_1 b_0$ and the multiplier by $a_2 a_1 a_0$ since $K = 4$ & $J = 3$. The logic diagram of the multiplier is shown below.



Division Algorithms :-

Division of two fixed-point binary numbers in signed-magnitude representation.

H/w implementation for signed-magnitude data :-

The h/w for implementing the division operation is identical to that required for multiplication

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

Divide Overflow :-

The division operation may result in a quotient with an overflow.

When the dividend is twice as long as the divisor, the condition for overflow can be stated as follows:

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

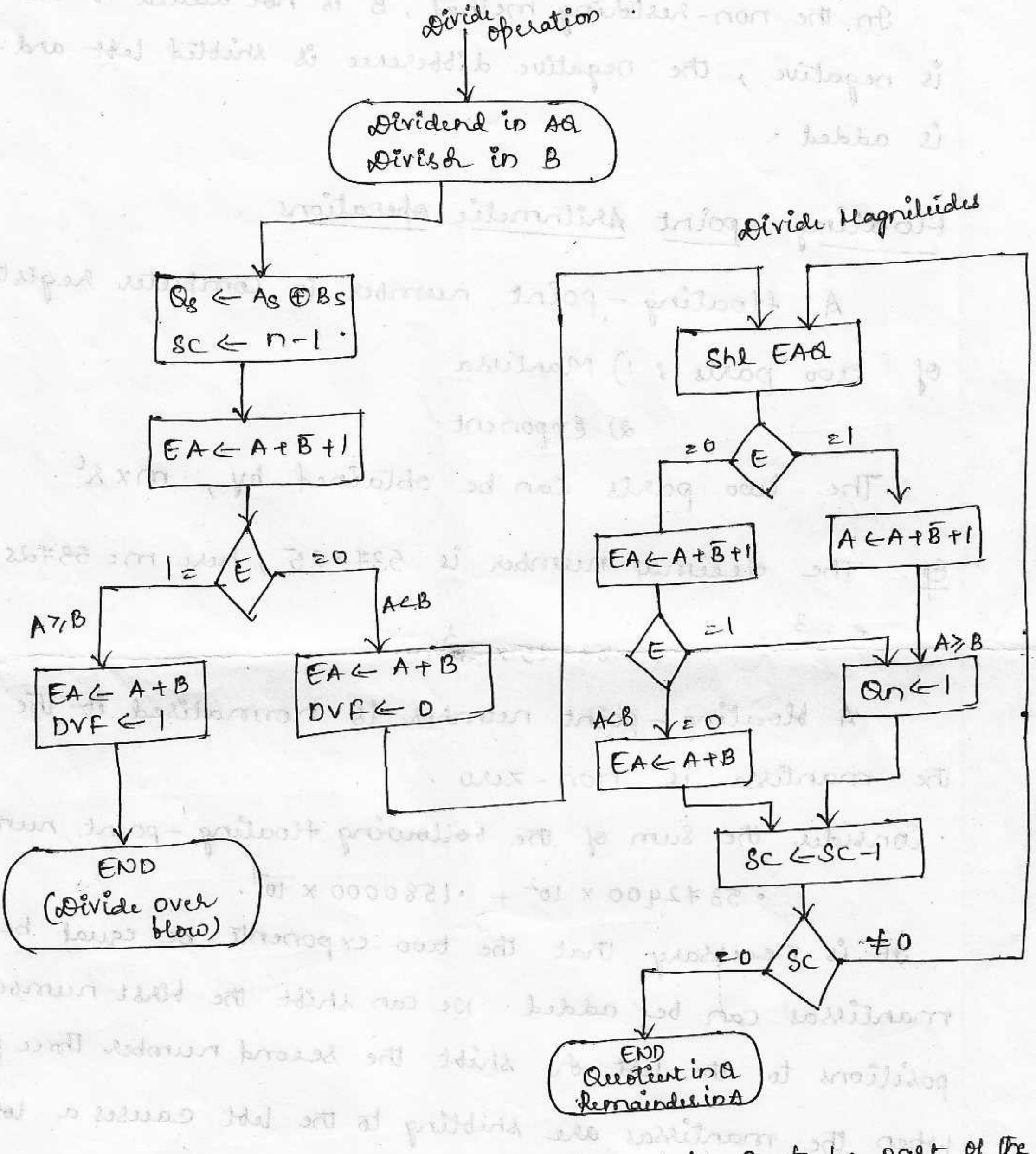
Another problem associated with division is the fact that a division by zero must be avoided.

Overflow condition is usually detected when a special flip-flop is set. We call it a divide-overflow flip-flop and label it DVF.

The best way to avoide a divide overflow is to use floating-point data.

Hardware Algorithm :-

The h/w divide algorithm is shown in the blockchart.



The sign of the result is transferred to Q_s to be part of the quotient.

Other Algorithms

The h/w method which described is called the restoring method. The two other methods are the comparison method and the non restoring method.

When two numbers are added, the sum may contain an overflow digit. An overflow can be corrected by shifting the sum once to the right & incrementing the exponent.

We can subtract two numbers in the following way.

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

A floating-point number that has a '0' in the MSB of the Mantissa is said to have an underflow.

Floating-point multiplication and division don't require an alignment of the mantissas. The product can be formed by multiplying the two mantissas & adding the exponents. Division is accomplished by dividing the mantissa and subtracting the exponents.

The exponent may be represented in any one of three representations: Signed-Magnitude, Signed-2's Complement, or Signed-1's Complement.

A fourth representation is a biased exponent. 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.

for eg, exponent that ranges from -50 to 49. Internally, it is represented by two digits (with out a sign) by adding to it a bias of 50. The exponent register contains the number e+50 where 'e' is the actual exponent.

The advantage of biased exponents is that they contai only positive numbers. Another advantage is that the smallest possible biased exponent contains all zero.

Addition & Subtraction :-

The algorithm can be divided into four parts

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

Multiplication :-

The multiplication algorithm can be divided into four parts

1) check for zeros
2) Add the exponents
3) Multiply the mantissas
4) Normalize the product.

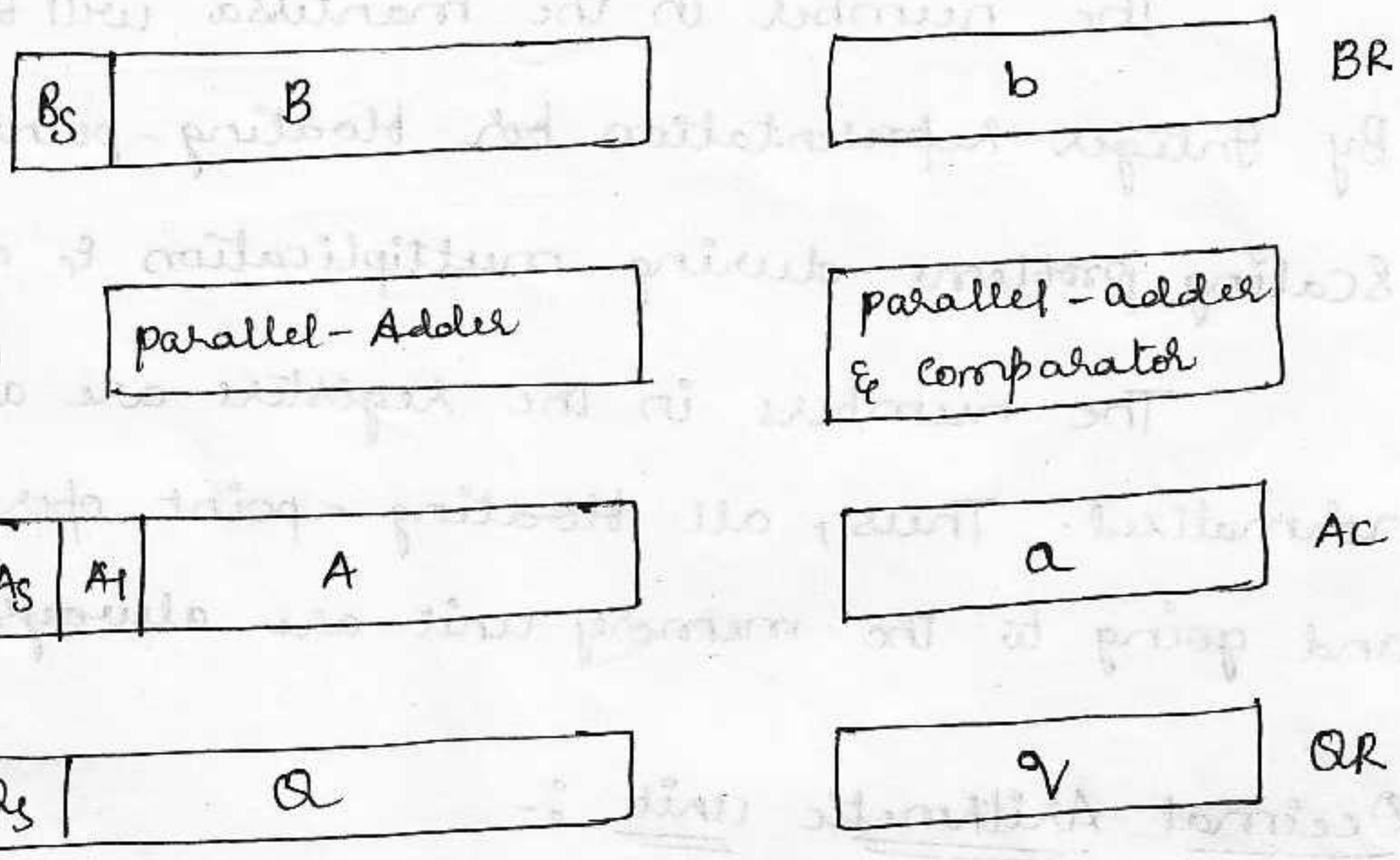
Division :-

The division algorithm can be divided into five parts:

1) check for zeros
2) Initialize the registers & evaluate the sign.
3) Align the dividend
4) Subtract the exponents.
5) Divide the mantissas.

Register configuration :-

The register configuration for floating-point operations as shown in fig.



There are three registers BR, AC & QR. Each register is subdivided into two parts. The mantissa part & the exponent part.

The Mantissa part has the same uppercase letter symbols as in fixed-point representation. The exponent part uses the corresponding lowercase letter symbol. Each floating-point number has a mantissa in signed-magnitude representation and a biased exponent. Thus AC has mantissa whose sign is in As and a magnitude is in A. The exponent is represented in 'a'. The diagram shows the MSB of A, is labeled by A1 whose value must be '1' to get normalised. In the same way, Register BR is subdivided into Bs, b & B and QR into Qs, Q, q.

A parallel-adder adds the two mantissas and transfers the sum into A and carry to 'E'. A separate parallel-adder is used for the two exponents. It ~~does~~, they don't have a distinct sign bit, but are represented as a biased positive quantity.

The exponents are also connected to a magnitude comparator that provides three binary o/p's to indicate relative magnitude.

The number in the mantissa will be taken as a fraction. By Integer representation for floating-point they may cause scaling problems during multiplication & division.

The numbers in the registers are assumed to be initially normalized. Thus, all floating-point operands coming from and going to the memory unit are always normalized.

Decimal Arithmetic unit :-

To perform arithmetic operations with decimal data, it is necessary to convert the i/p decimal numbers to binary, it is necessary perform all calculations with binary numbers and to convert the results into decimal. It is very efficient Method. The decimal numbers are then applied to a decimal arithmetic unit to executing decimal arithmetic micro operations.

Many Computers have h/w for arithmetic calculations with both binary & decimal data.

A decimal arithmetic unit is a digital function that performs decimal micro operations. A single-stage decimal arithmetic unit consists of nine binary i/p variables & five binary o/p variables, since a minimum of four bits is required to represent each coded decimal digit.

BCD Adder :-

In BCD, each i/p digit doesn't exceed 9, the o/p sum can't

be greater than 9+9+1 = 19, the '1' in the sum being an i/p carry.

For eg, we apply two BCD digits to a 4-bit binary adder. The adder will form the sum in binary & produce a result that may range from 0 to 19. These binary numbers are shown in below table and are labeled by symbols K, Z_8, Z_4, Z_2 & Z_1, K is the carry. The first column in the table lists the binary sums as they appear in the O/p of a 4-bit binary adders. The O/p sum of two decimal numbers must be represented in BCD. In the second column, the values of the first value can be converted to the correct BCD digit.

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

In the table, When the binary sum is equal to or less than 1001, the corresponding BCD number is identical, and there is no conversion is needed. When the binary is greater than 1001, than nonvalid BCD representation is obtained.

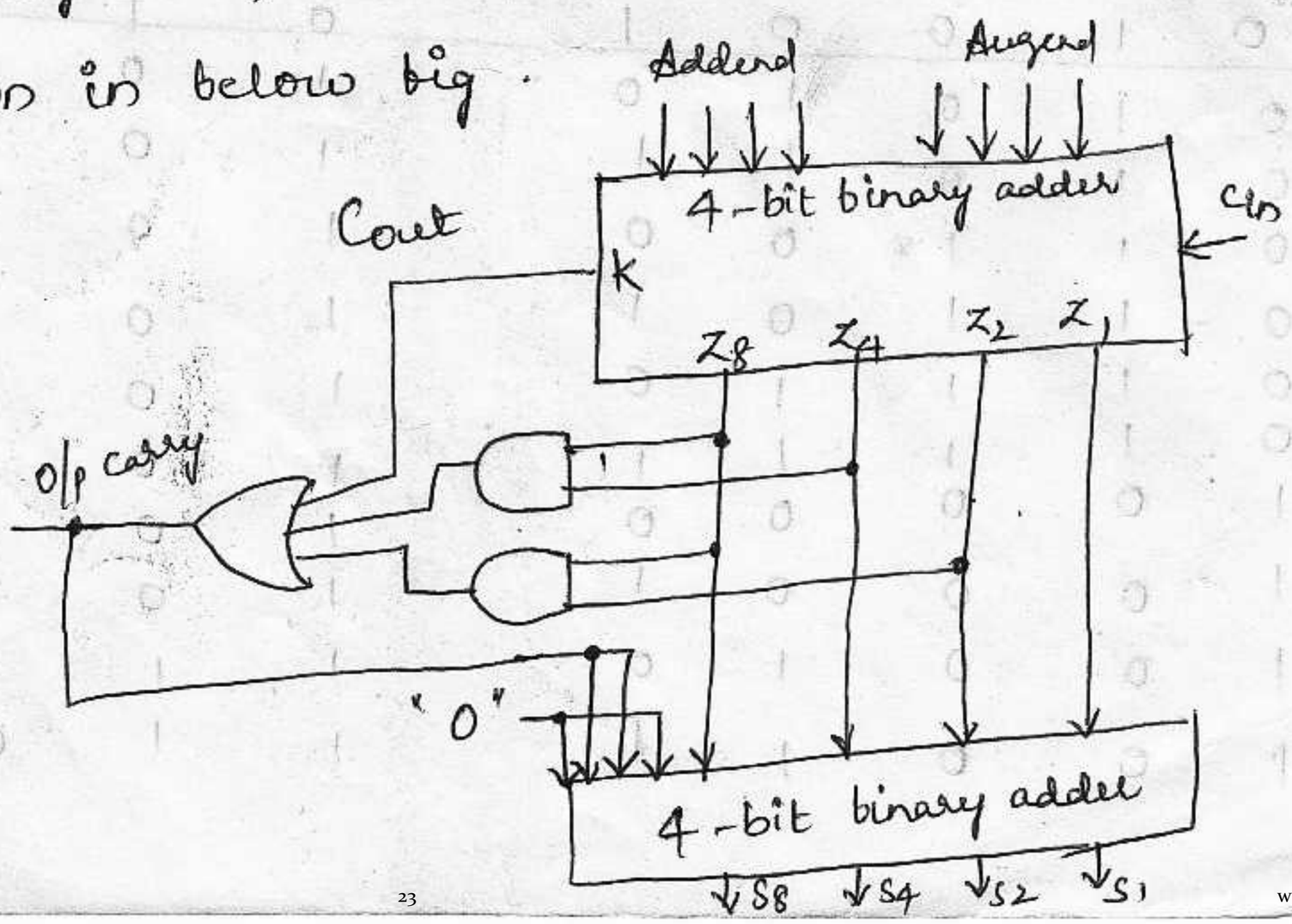
One method of adding decimal numbers in BCD would be to employ one 4 bit binary adder & perform the arithmetic operation one digit at a time. If the result is greater than or ~~greater than~~ equal to 1010, it corrected by adding 0110 to the binary sum. The second operation will automatically produce an o/p carry for the next pair of significant digits. The procedure is repeated until all decimal digits are added.

The Condition for a Correction and an o/p - 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 & produces a sum digit also in BCD. To add 0110 to the binary sum, we use a second 4-bit binary adder as shown in below fig.



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 it is equal to 1, binary 0110 is added to the binary sum through the bottom 4-bit binary adder. The o/p-carry generated from the bottom binary adder may be ignored, since it supplies information already available in the o/p-carry terminal.

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.

BCD Subtraction :-

BCD is not a self-complementing code, the 9's Complement cannot be obtained by complementing each bit in the code.

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.

1) binary 1010 (decimal 10) is added to each complemented digit and the carry discarded after each addition

2) binary 0110 (decimal 6) is added before the digit is complemented.

Eg: 1) 0111

Complement

1000

1010 (add)

0010

2) 0111

0111 (add)

0110

1101

0010 (complement)

Complementing each bit of a 4-bit binary number N is identical to the subtraction of the number from 1111 (decimal 15)

1) Adding the binary equivalent of decimal 10 gives

$$15 - N + 10 = 9 - N + 16$$

So, here 16 signifies the carry is discarded. So, the result is $9 - N$

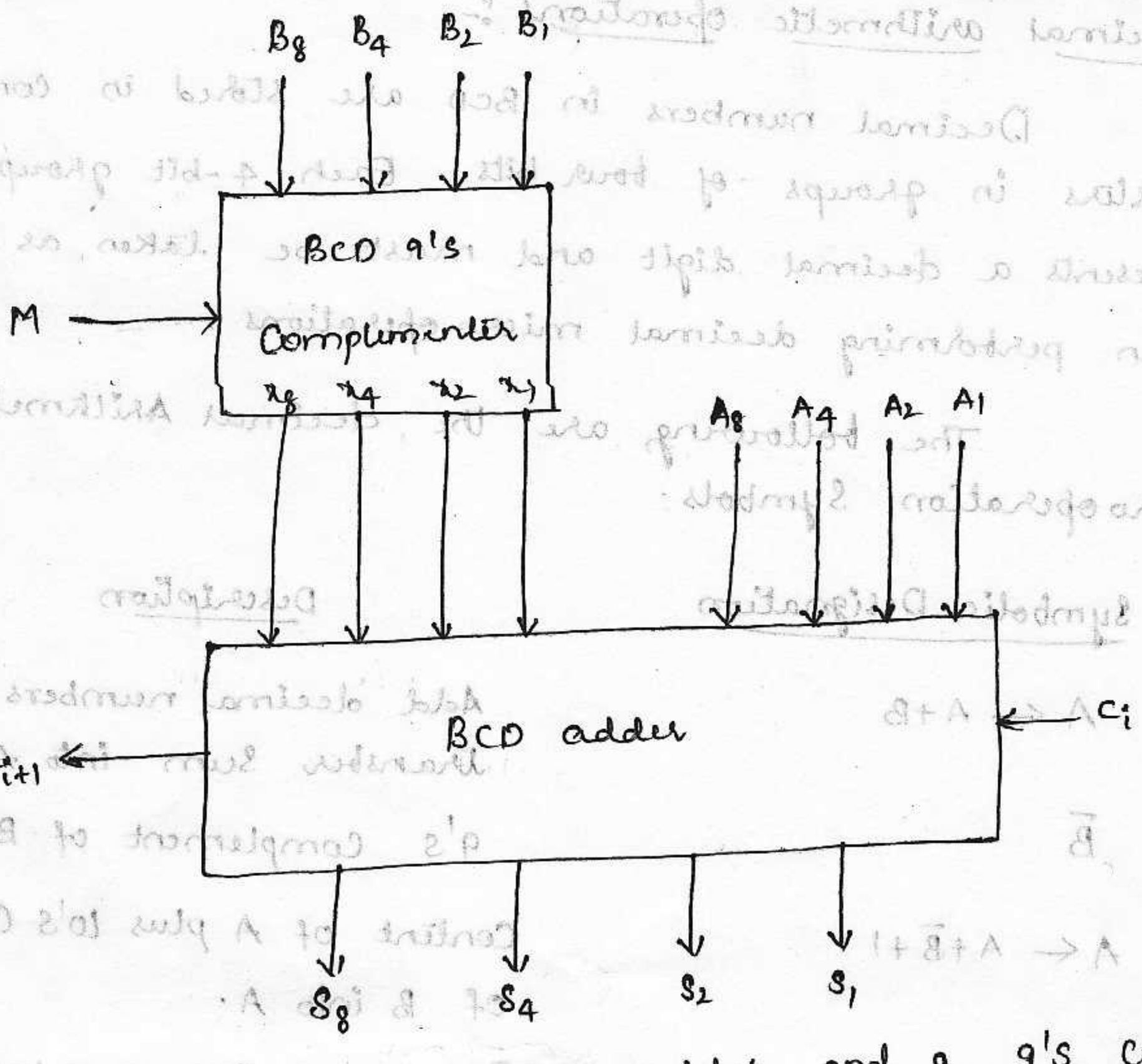
2) Adding the binary equivalent of decimal 6 gives

$$15 - (N + 6) = 9 - N.$$

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 BCD adder / subtractor.

Let the subtrahend (or addend) digit be denoted by the four binary variables B_8, B_4, B_2, B_1. Let M be a mode bit that controls the add / subtract operation.

Let the binary variables x_8, x_4, x_2, x_1 be the o/p's of the 9's complementer circuit. The below fig. shows the one stage of a decimal arithmetic unit.



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' o/p's form the sum of A and B.

With M=1, the 'S' o/p's form the sum of A plus the 9's complement of B.

The o/p carry C_{i+1} from one stage must be connected to the i/p carry C_i of the next higher-order stage.

The way to subtract the two decimal numbers is to let M=1 and apply a '1' to the i/p carry C_1 of the first stage. The o/p's 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.

Decimal arithmetic operations :-

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 micro operations.

The following are the decimal Arithmetic micro operation Symbols.

Symbolic Designation	Description
$A \leftarrow A + B$	Add decimal numbers and transfer sum into A.
$\bar{B}$	9's Complement of B.
$A \leftarrow A + \bar{B} + 1$	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

Incrementing / decrementing a register is the same for binary and decimal except that the binary counter goes through 16 states from 0000 to 1111, when incremented. The decimal counter goes through 10 states from 0000 to 1001 and back 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.

Eg: 3 4 2 1

0011 0100 0010 0001

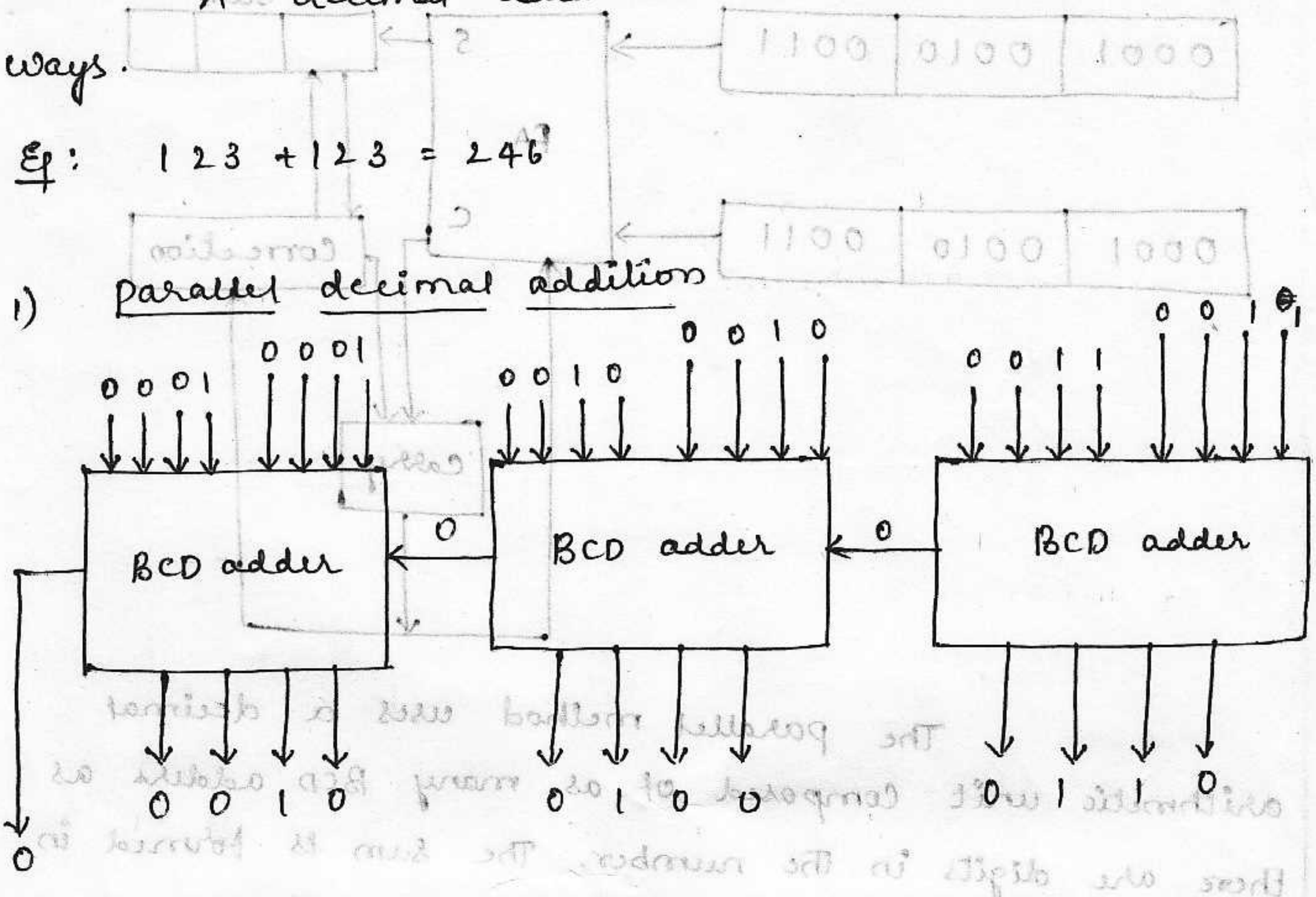
dshr 0000 0011 0100 0010

Addition and Subtraction :-

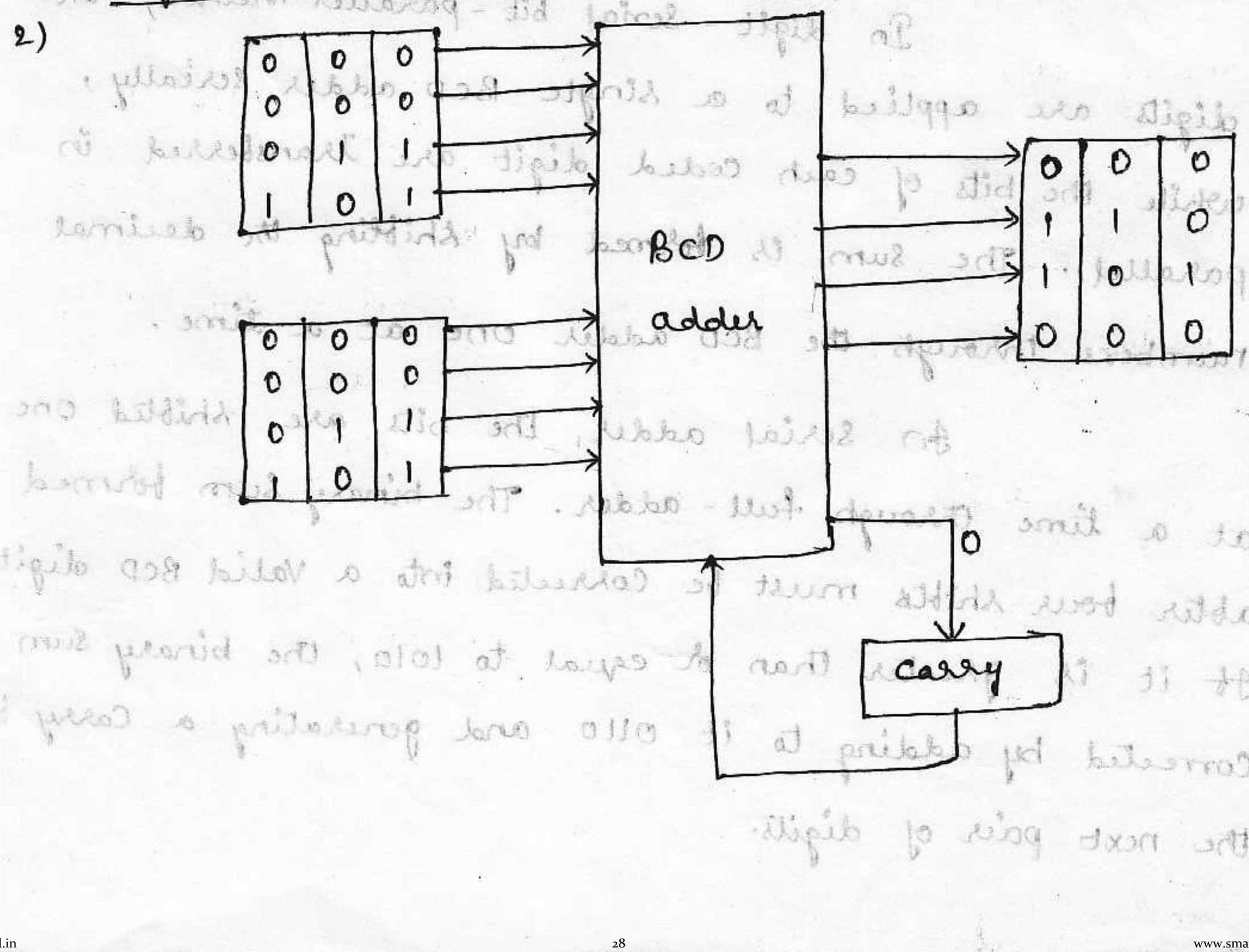
A decimal data can be added in three different ways.

Eg: 123 + 123 = 246

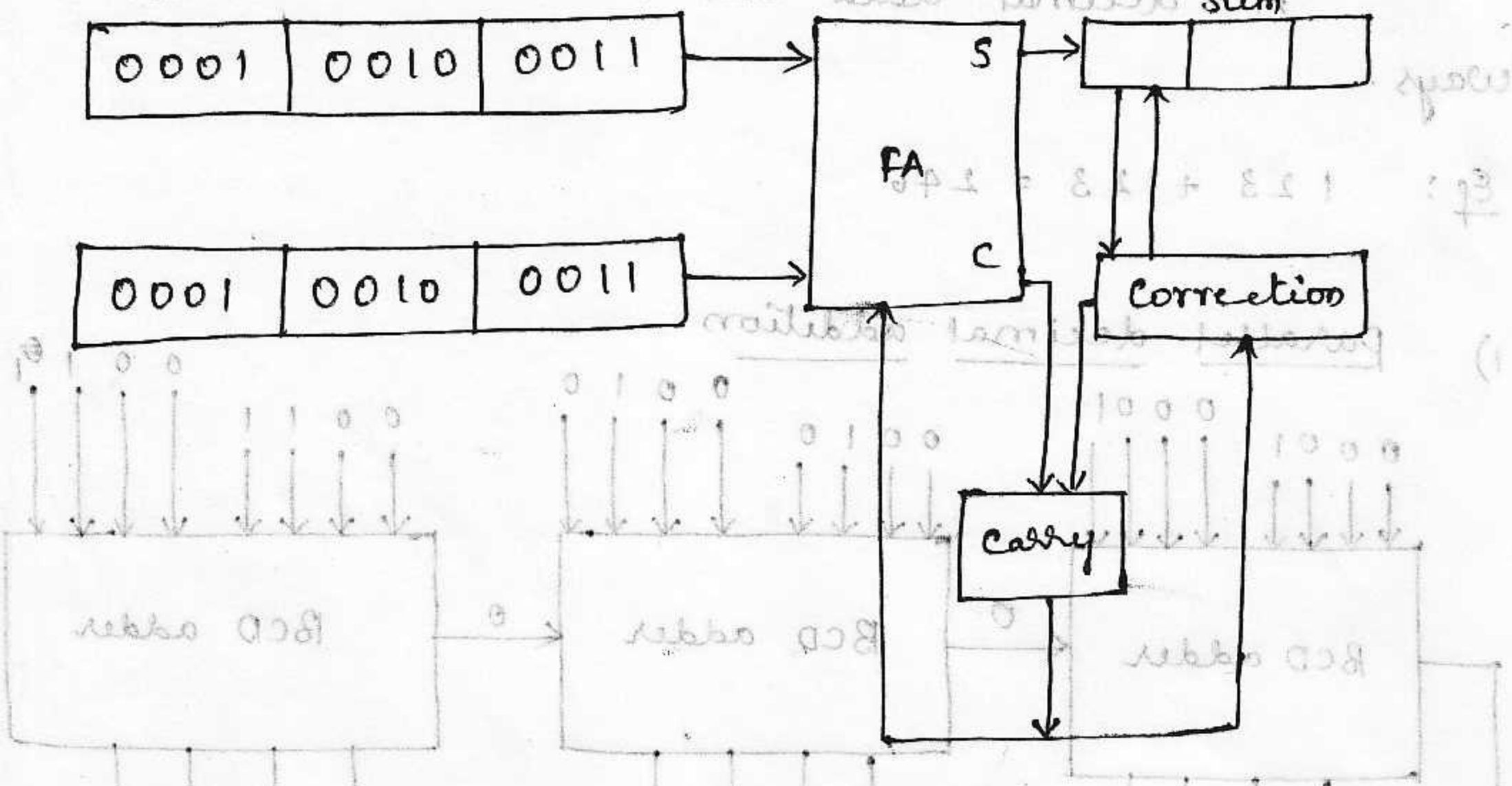
1) Parallel decimal addition



2) Digit - Serial, bit parallel decimal addition



3) Serial adder



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 micro operation.

In 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.

In serial adder, the bits are shifted one at a time through full - adder. The binary sum formed after four shifts must be corrected into a valid BCD digit. 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.

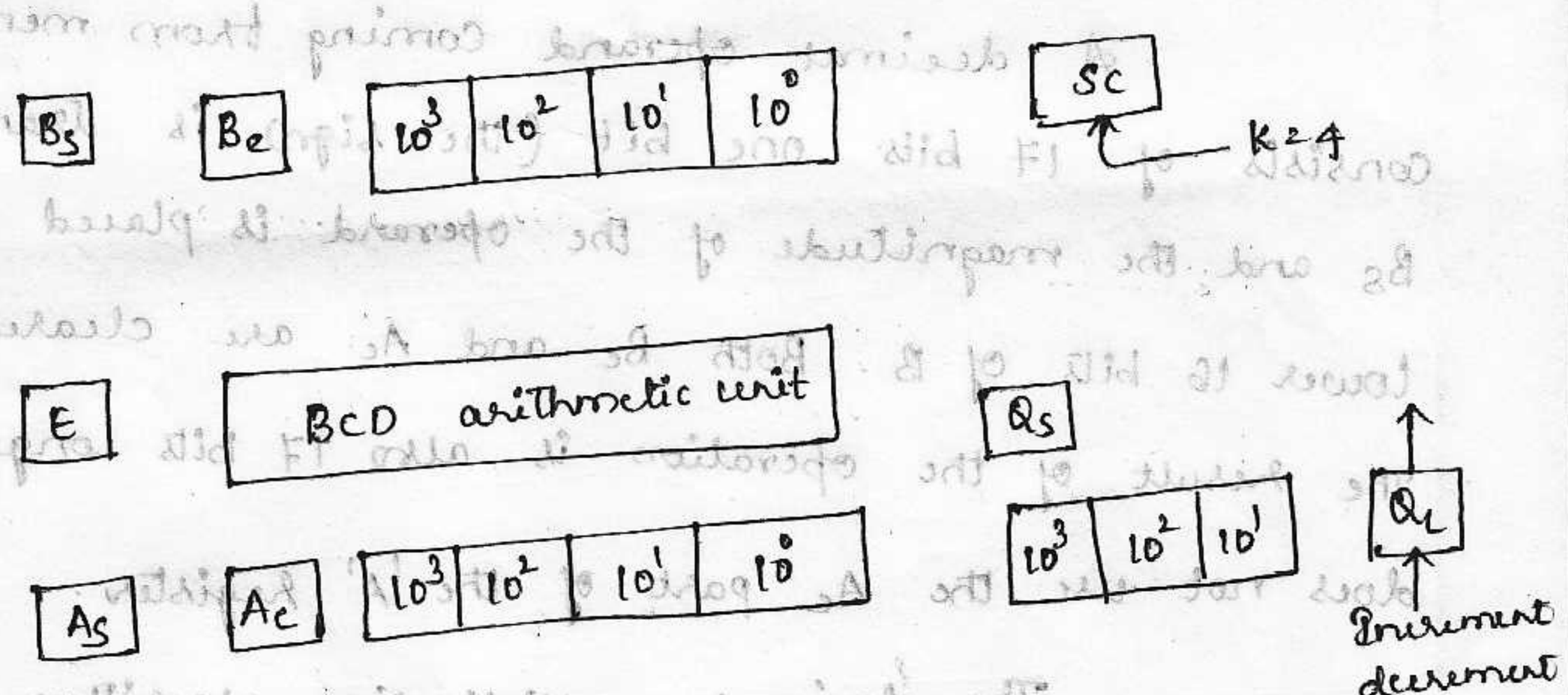
The parallel method is fast but requires a large number of adders.

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 serial method requires a minimum amount of equipment but is very slow.

Multiplication :-

The registers configuration for the decimal multiplication is shown in fig.



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 & Q each having a corresponding sign flip-flop A_S, B_S & Q_S.

Registers A and B have four more bits designated by A_e & 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 the flip flop 'E'.

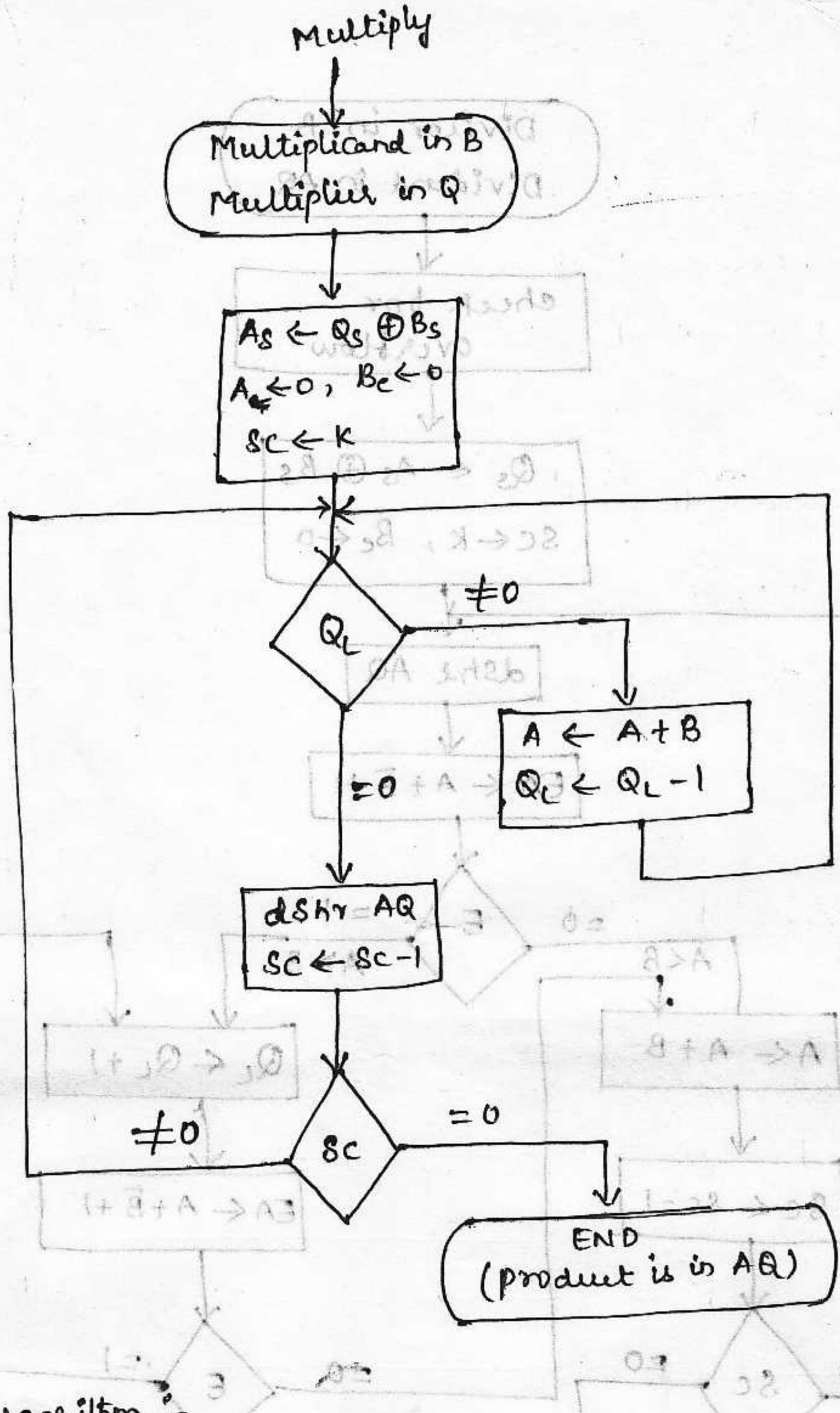
The purpose of digit A_e is to accommodate an overflow while adding the multiplicand to the partial product during multiplication.

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 below.



Division algorithm :-

In the restoring division method, the divisior is subtracted from the dividend & partial remainder as many times as necessary until a negative remainder results. The correct remainder is then restored by adding the divisior. The digit in the quotient reflects the no. of subtractions, upto but excluding the one that caused the negative difference.

The decimal division algorithm is shown below.