RGPVNOTES.IN

Program : **B.Tech**

Subject Name: **Computer Organization and Architecture**

Subject Code: **CS-404**

Semester: **4th**

UNIT – 3

Syllabus :Computer Arithmetic: Addition and Subtraction, Two's Compliment Representation, Signed Addition and Subtraction, Multiplication and division, Booths Algorithm, Division Operation, Floating Point Arithmetic Operation, Design of Arithmetic unit.

INTRODUCTION

- Based on the number system two basic data types are implemented in the computer system: fixed point numbers and floating point numbers.
- Representing numbers in such data types is commonly known as fixed point representation and floating point representation, respectively.
- In binary number system, a number can be represented as an integer or a fraction.
- Depending on the design, the hardware can interpret number as an integer or fraction.
- The radix point is never explicitly specified It is implicited in the design and the hardware interprets it accordingly.
- In integer number radix point is fixed and assumed to be to the right of the right most digits.
- As radix point is fixed, the number system is referred to as fixed point number system.
- With fixed point number system we can represent positive or negative integer numbers.
- Floating point number system allows the representation of numbers having both integer part and fractional part.

ADDITION AND SUBTRACTION OF SIGNED NUMBERS

We can relate addition and subtraction operations of numbers by the following Relationship:

$$(\pm A) - (+B) = (\pm A) + (-B) \text{ and } (\pm A) - (-B) = (\pm A) + (+B)$$

Therefore, we can change subtraction operation to an addition operation by changing the sign of the subtrahend.

1's Complement Representation

The 1's complement of a binary number is the number that results when we change all 1's to zeros and the zeros to ones.
Find 1's complement of $(11000100)_2$.

Solution -

1 1 0 0 0 1 0 0
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
0 0 1 1 1 0 1 1

2's Complement Representation

The 2's complement is the binary number that results when we add 1 to the 1's Complement. It is given as 2's complement = **1's complement + 1**
The 2's complement form' is used to represent negative numbers.

Find 2's complement of $(11000100)_2$

Solution – 1	1	0	0	0	1	0	0	
	0	1	1	1	0	1	1	1's compliment
0	0	1	1	1	0	1	1	1's compliment
+							1	2's compliment
	0	0	1	1	1	1	0	0

Subtraction of Binary Numbers using 2's Complement Method

In a 2's complement subtraction, negative number is represented in the 2's complement form and actual addition is performed to get the desired result. For example, operation A-B is performed using following steps:

1. Take 2's complement of B.
2. Result A + 2's complement of B.
3. If carry is generated then the result is positive and in the true form. In this case, carry is ignored.
4. If carry is not generated then the result is negative and in the 2's complements form.

Addition / Subtraction Logic Unit

Figure 1 shows hardware to implement integer addition and subtraction. It consists of n-bit adder, 2's complement circuit, overflow detector logic circuit and AVF (overflow flag). Number a and number b are the two inputs for n-bit adder. For subtraction, the subtrahend (number from B register) is converted into its 2's complement form by making Add/ Subtract control signal to the logic one. When Add/Subtract control signal is one, all bits of number b are complemented and carry zero (C0) is set to one. Therefore n-bit adder gives result as R = a + b(bar) + 1, where b (bar) +1 represents 2's complement of number b.

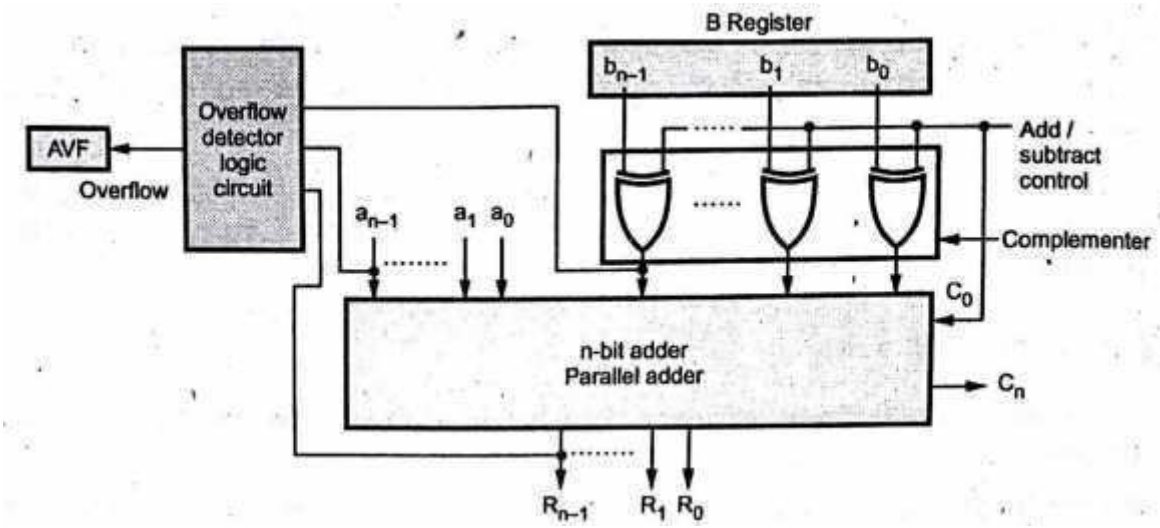


Figure 1: Hardware for Integer addition and subtraction

Overflow In Integer Arithmetic

Assuming numbers are in 4 bit for addition and subtraction.

Case 1: Both numbers positive

```
  1 1 1    carry bit
0 1 1 1    (+ 7)
0 0 1 1    (+ 3)
-----------
1 0 1 0    Result: 2's complement of 6
```

Result is -6 ; it is wrong due to overflow.

Case 2: Both numbers negative

```
    1          Carry
1 0 1 1    2's complement of 5, i.e. (- 5)
1 1 0 0    2's complement of 4, i.e. (- 4)
1 0 1 1 1  (+ 7)
```

- Result is + 7 ; it is wrong due to overflow,
- When adding signed a, numbers, a carry bit beyond the end of the word does not serve as the overflow indicator.
- If we add then numbers +7 and + 3 in a 4-bit adder, the output is 1010, which is the code of -6, a wrong result.
- In this case, carry bit from the MSB position is 0. Similarly, if we add -5 and -4, we get output = + 7, another error.
- In this case carry bit from the MSB position is 1.
- One thing we can surely say that, the addition of numbers with different signs cannot cause overflow, because the absolute value of the sum is always smaller than the absolute value of one of the two operands.

From above discussion we can conclude following points:

1. Overflow can occur only when adding two numbers that have the same sign.
2. The carry bit from the MSB position is not a sufficient indicator of overflow when adding signed numbers.
3. When both operands a and b have the same sign, an overflow occurs when the sign of result does not agree with the signs of a and b. The logical expression to detect overflow can be given as

Overflow = $a_{n-1}\, b_{n-1}\, \overline{R_{n-1}} + \overline{a_{n-1}}\, \overline{b_{n-1}}\, R_{n-1}$

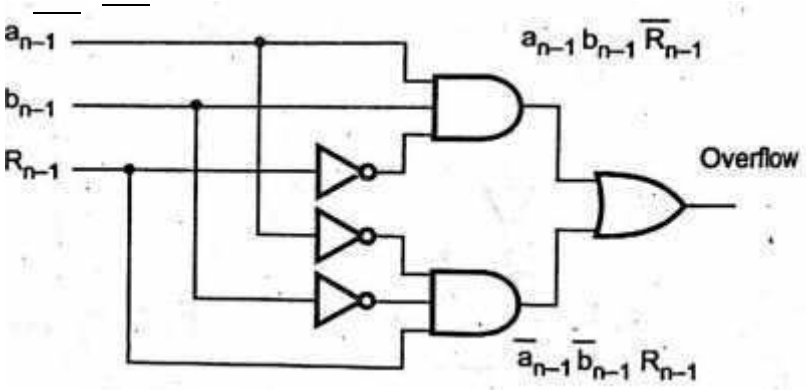


Figure 2: Overflow detector logic circuit

ADDITION AND SUBTRACTION ALGO & FLOWCHART

There are three ways of representing negative fixed-point binary numbers:

1. Signed magnitude
2. Signed- 1's complement
3. Signed-2's complement.

Most computers use the signed-2's complement representation when performing arithmetic operation's with integers. For floating-point operations, most computers use the signed-magnitude representation for the mantissa.

Addition and Subtraction with Signed Magnitude Data

The representation of numbers in signed-magnitude is familiar because it is used in everyday arithmetic calculations. 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 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.

Table 1: Addition and Subtraction of Signed Magnitude numbers

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

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.

Consider now the hardware implementation of the algorithms above.

- First, a parallel adder is needed to perform the micro operation 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 micro operations A-B and B-A.

The sign relationship can be determined from an exclusive - OR gate with A_S and B_S as inputs.

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 complementor.

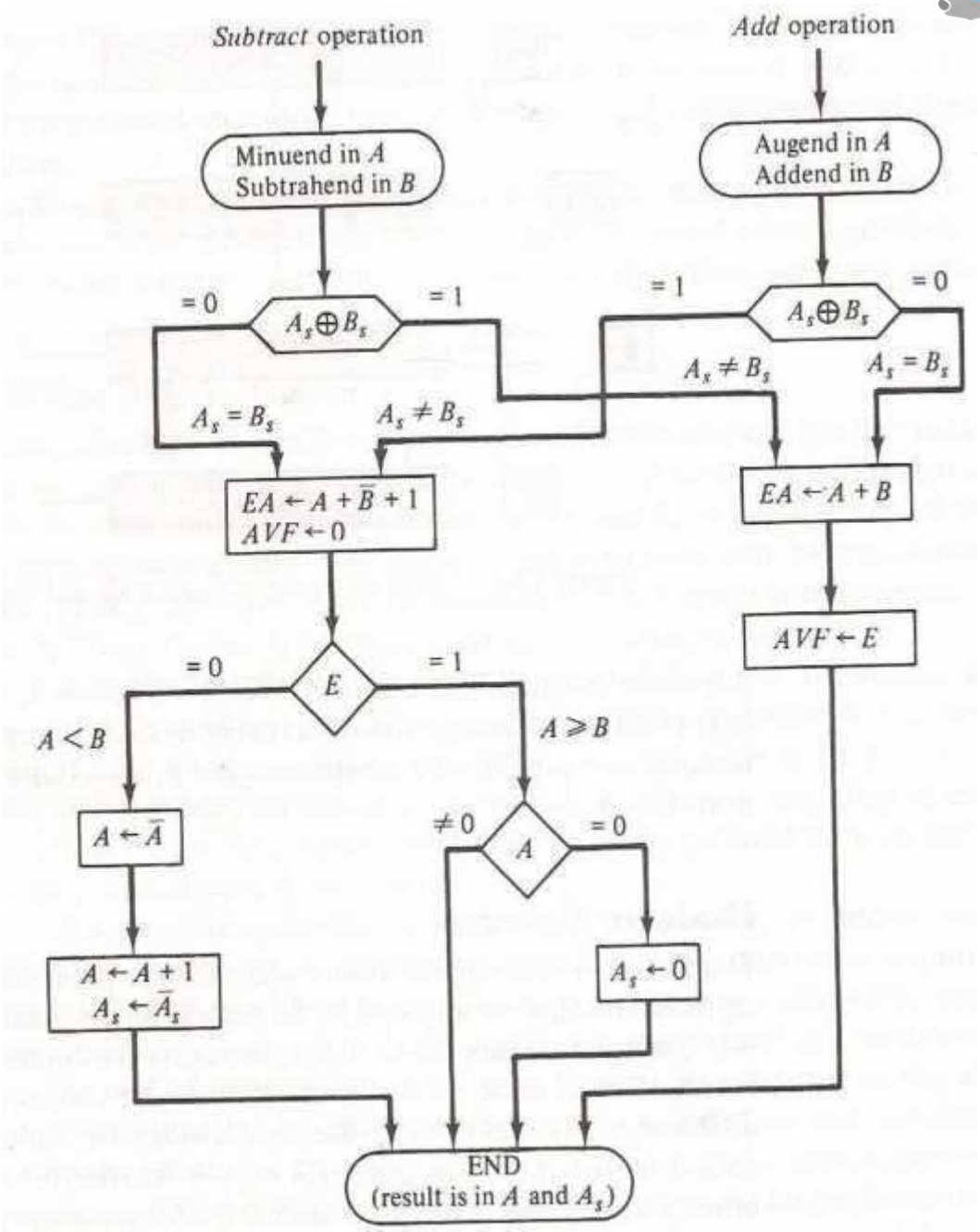


Figure 3: Flowchart of Add & Subtract Operation

The flowchart for the hardware algorithm is presented in Figure 3. 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 then signs are different.

For an add operation, identical signs indicate that the magnitudes be added. For a subtract operation, different signs indicate that the magnitudes be added. The magnitudes are added with a micro operation EA<- 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 to1. 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 1in E indicates that A>=B and the number in A is the correct result. If this number is zero, the sign As 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 micro operation A<-Ã + 1. Here, we assume that the A register has circuit for micro operations complement and increment, so the 2's complement is obtained from these two micro operations.

In other paths of the flowchart, the sign of the result is the same as the sign of A, so no change in AS 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 AS to obtain the correct sign. The final result is found in register A and its sign in AS.

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. The left most 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 are represented as 00100001 and -33 as 11011111. Note that 11011111 is the 2's complement of 00100001 and vice versa.

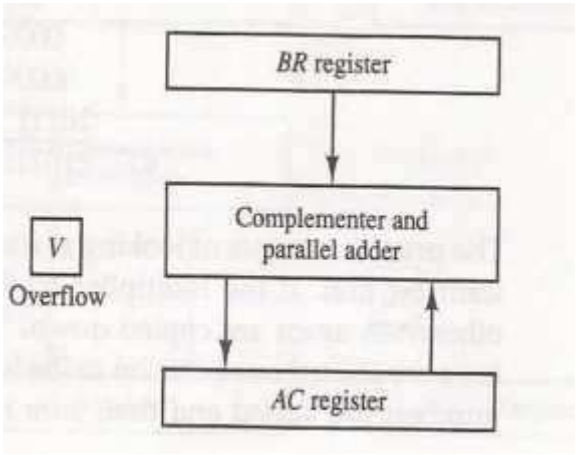


Figure 4: Hardware for signed 2'scomplement addition and subtraction

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 digit, we say that an overflow occurred. An overflow can be detected by inspecting the last two carries out of the addition.

The algorithm for adding and subtracting two binary numbers in signed- 2's complement representation is shown in the flowchart of Figure 5. 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 l, 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.

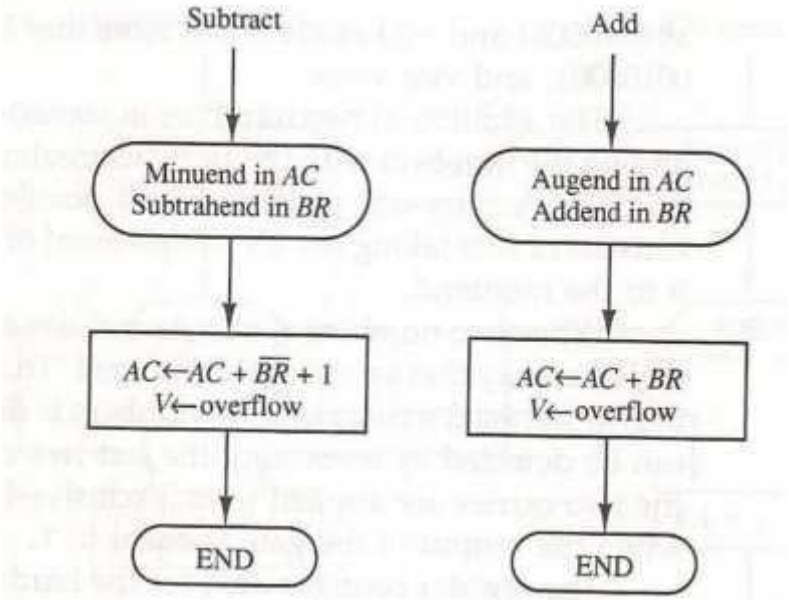


Figure 5: Algorithm for adding and subtracting numbers in signed-2's complement form representation

An overflow must be checked during this operation because the two numbers added could have same sign. The programmer must realize that if an overflow occurs, there will be an erroneous result in the AC register. 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.

MULTIPLICATION ALGO. & FLOWCHART

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

```
 23      10111    Multiplicand
 19    × 10011    Multiplier
        10111
       10111
      00000   +
     00000
    10111
437 110110101    Product
```

The process consists of looking at successive bits of the multiplier, LSB first. If the multiplier bit is a 1 multiplicand is copied down otherwise, zeros are copied down. The numbers copied down in successive lines are shifted one position to the left from the previous number. Finally, the numbers are added and their sum forms the product. The sign of the product is determined from the sign of the multiplicand and multiplier. If they are alike, the sign of the product is positive. If they are unlike, the sign of the product is negative.

As shown in Figure 6: initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in B_S and Q_S respectively. The signs are compared, and both A and Q are set to correspond to the sign of the product since a double length product will be stored in registers A and Q. Registers A and E are cleared and the sequence counter SC is set to a number equal to the number of bits of the multiplier.

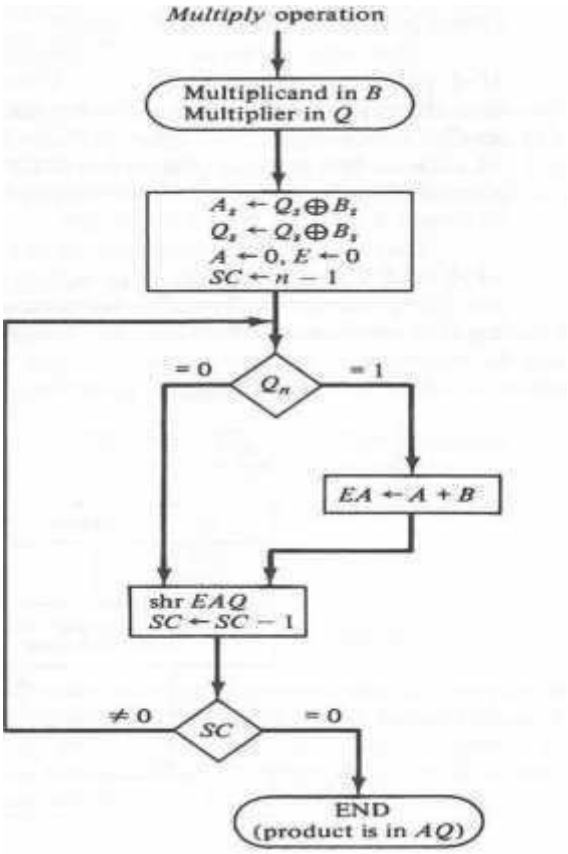


Figure 6: Flowchart of the hardware multiply algorithm.

We are assuming here that operands are transferred to registers from a memory unit that has words of n bits. Since operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of n-1 bits.

After the initialization, the low order bit of the multiplier in Q 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 from the new partial product. The sequence counter is decremented by 1 and its new value checked. If it is not equal to zero, the process is repeated and a new partial product is formed. The process stops when SC= 0. Note that the partial product formed in A is shifted into Q one bit at a time and eventually replaces the multiplier. The final product is available in both A and Q, with A holding the most significant bits and Q holding the least significant bits.

Table 2: Binary Multiplication example

Multiplicand B = 10111	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
Q_n = 1; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
Q_n = 1; add B		10111		
Second partial product	1	00010		
Shift right EAQ	0	10001	01100	011
Q_n = 0; shift right EAQ	0	01000	10110	010
Q_n = 0; shift right EAQ	0	00100	01011	001
Q_n = 1; add B		10111		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000
Final product in AQ = 0110110101				

Booth Multiplication Algorithm

Booth algorithm gives a procedure for multiplying binary integers in signed 2'scomplement representation.

It operates on the fact that strings of 0's in the multiplier require no addition but just shifting and a string of 1's in the multiplier from bit weight 2K to weight 2M can be treated as 2K+ l-2M.

Booth algorithm requires examination of the multiplier bits and shifting of the partial product. Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial product, or left unchanged according to the following rules:

1. The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier.
2. The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous 1) in a string of 0' s in the multiplier.
3. The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

Table 3: Example of booth Multiplication

$Q_n Q_{n+1}$	$BR = 10111$ $\overline{BR} + 1 = 01001$	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	Subtract BR	01001			
		01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	10111			
		11001			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract BR	01001			
		00111			
	ashr	00011	10101	1	000

The hardware implementation of Booth algorithm requires the register configuration shown in Figure 7. This is similar to Figure 6, above (multiply) except that the sign bits are not separated from the rest of the register. To show this difference, we rename registers A, B, and Q as AC, BR, and QR, If the two bits are equal to 10, it means that the first 1 in a string of 1's has been encountered. This requires a subtraction of the multiplicand from the partial product in AC. If the two bits are equal to 01 it means that the first 0 in a string of 0 's has been encountered. This requires the addition of the multiplicand to the partial product in AC . When the two bits are equal, the partial product does not change. An overflow cannot occur because the addition and subtraction of the multiplicand follow each other. As a consequence, the two numbers that are added always have opposite sign, a condition that excludes an overflow. The next step is to shift right the partial product and the multiplier (including bit Qn +1). This is an arithmetic shift right (ashr) operation which shifts AC and QR to the right and leaves the sign bit in AC unchanged. The sequence counter is decremented and the computational loop is repeated n times.

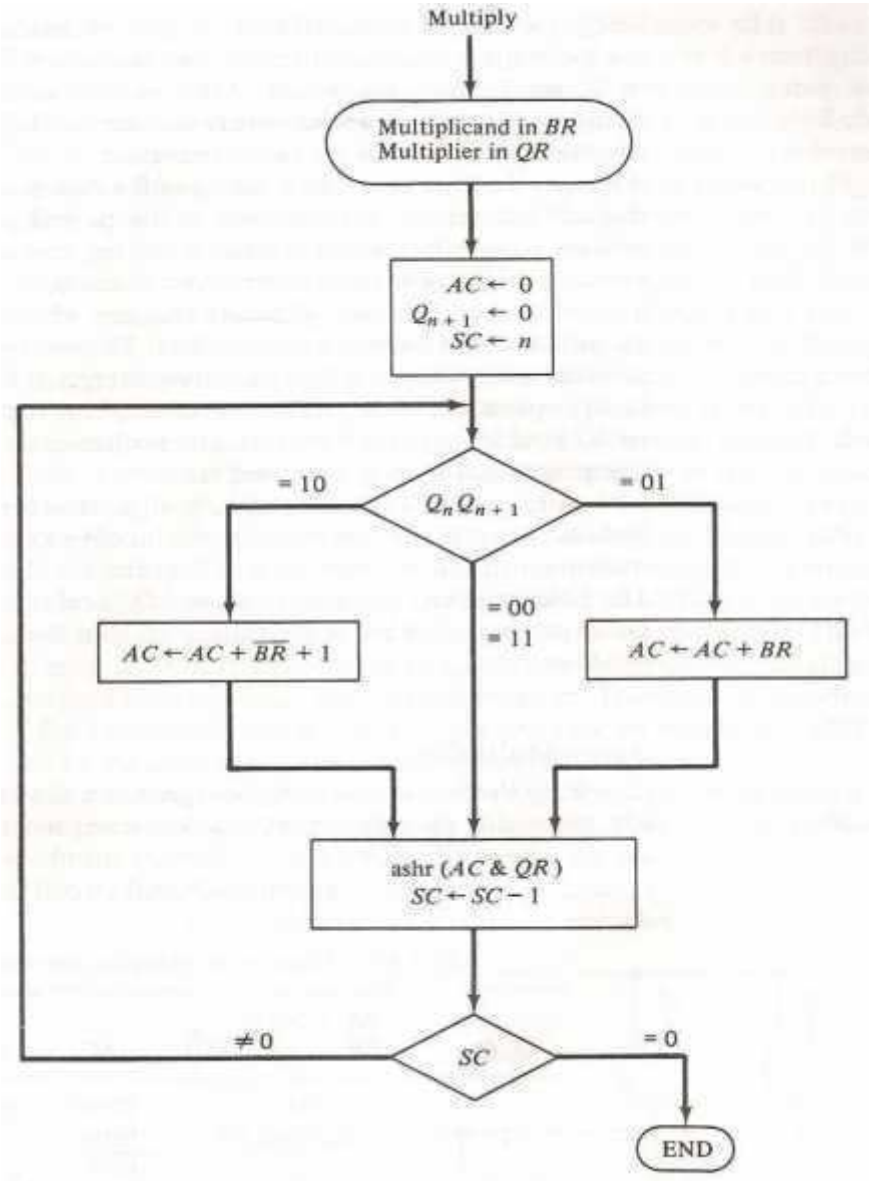


Figure 7: Flowchart for Booth Algorithm

DIVISION ALGO. & FLOWCHART

When the division is implemented in a digital computer, it is convenient to change the process slightly. Instead of shifting the divisor to the right, the dividend, or partial remainder, is shifted to the left, thus leaving the two numbers in the required relative position. Subtraction may be achieved by adding A to the 2's complement of B. The information about the relative magnitudes is then available from the end carry.

The hardware for implementing the division operation is identical to that required for multiplication and consists of the component shown above multiplication. In Figure 8 register EAQ is now shifted to the left with 0 inserted into Q_n and the previous value of E lost

The divisor is stored in the B register and the double length dividend is stored in registers A and Q .The dividend is shifted to the left and the divisor is subtracted by adding its 2's complement value. The information about the relative magnitude is available in E. If E=1 it signifies that A>=B. A quotient bit 1 is inserted into Q_n and the partial remainder is shifted to the left to repeat the process. If E= 0, it signifies that A< B so the quotient in Q_n.

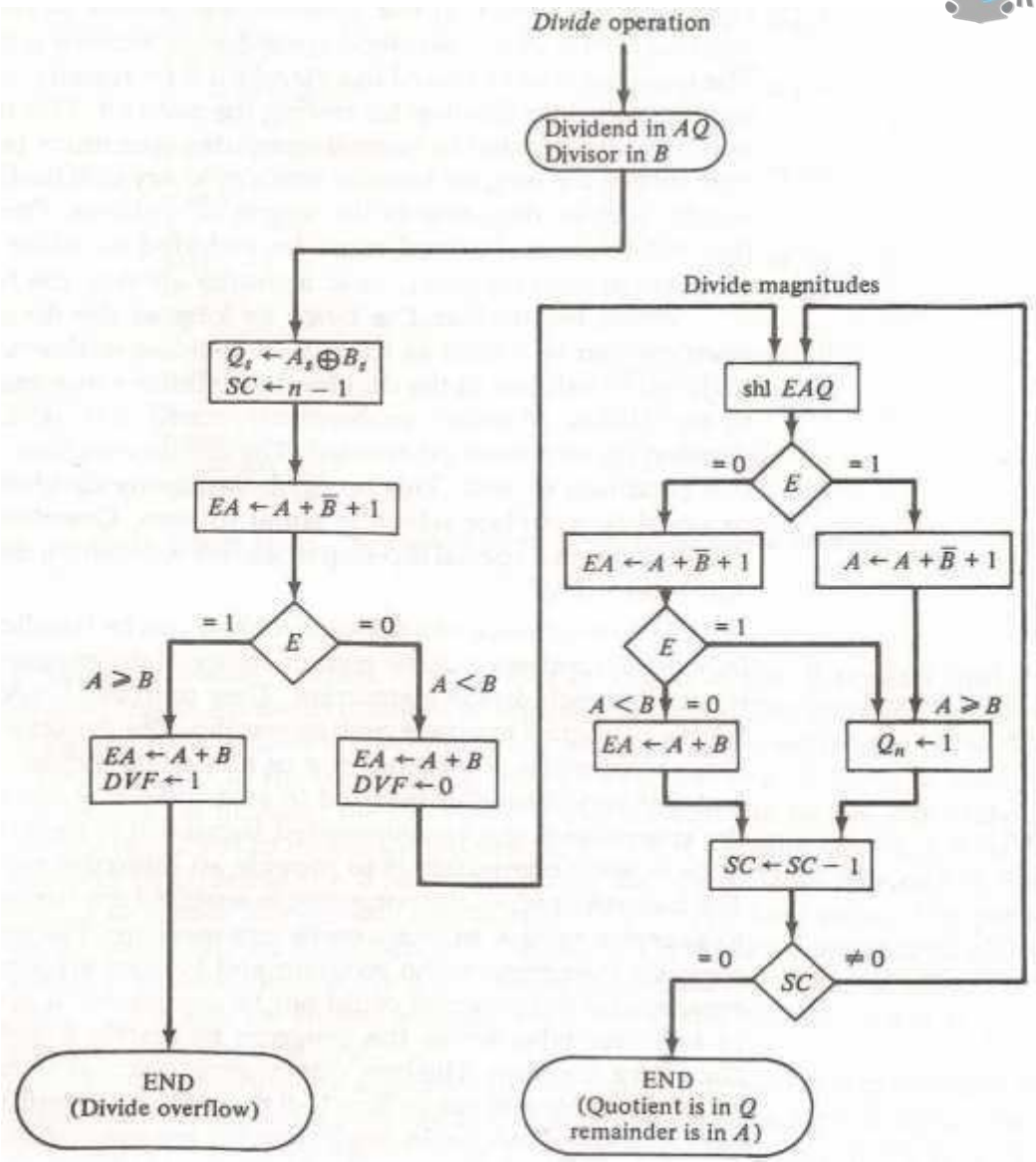


Figure 8: Flowchart for Division Algorithm

The hardware divide algorithm is shown in the flowchart of Figure 8.

- The dividend is in A and Q and the divisor in B.
- The sign of the result is transferred into QS to be part of the quotient.
- A constant is set into the sequence counter SC to specify the number of bits in the quotient.
- As in multiplication, we assume that operands are transferred to registers from a memory unit that has words of n bits.
- Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of n-1 bits.
- A divide overflow condition is tested by subtracting the divisor in B from half of the bits of the dividend stored in A.
- If A > B, the divide overflow flip-flop DVF is set and the operation is terminated prematurely.
- If A <B, no divide overflow occurs so the value of the dividend is restored by adding B to A.

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

	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\overline{B} + 1$		01111		
$E = 1$	1	01011		
Set $Q_n = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00101		
Set $Q_n = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	11001	00110	
Add B		10001		
Restore remainder	1	01010		2
shl EAQ	0	10100	01100	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00011		
Set $Q_n = 1$	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A:		00110		
Quotient in Q:			11010	

Figure 9: Division Algorithm Example

FLOATING POINT REPRESENTATION

To accommodate very large integers and very small fractions, a computer must be able to represent numbers and operate on them in such a way that the position of the binary point is variable and is automatically adjusted as computation proceeds.

In this case, the binary point is said to float, and the numbers are called floating- point numbers. The floating point representation has three fields: sign, significant digits and exponent.

To represent the number 111101.1000110 in floating point format, first binary point is shifted to right of the first bit and the number is multiplied by the correct scaling factor to get the same value. The number is said to be in the normalized form and is given as shown in below format.

$$111101.1000110 \text{ ---} 1.\ 11101100110 \times 2^{5}$$

5 <----------Exponent

Significant digits Scaling Factor

Number represented in normalized form

It is important to note that the base in the scaling factor is fixed 2. The string of the significant digits is commonly known as mantissa. In the above example

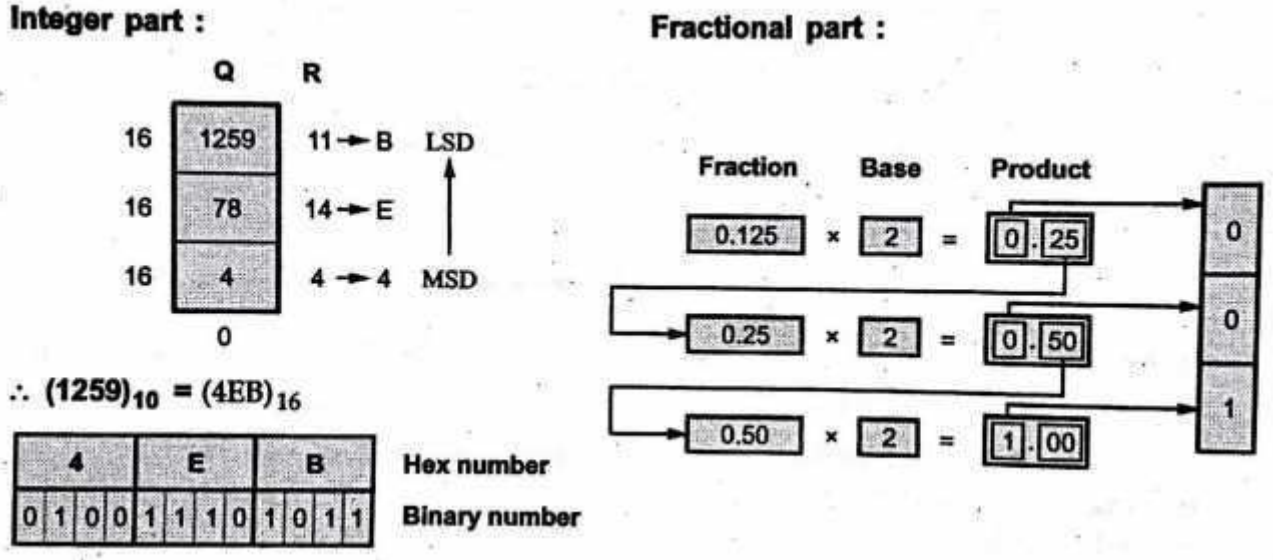
Sign = 0 Mantissa = 11101100110 Exponent = 5

In floating point numbers, bias value is added to the true exponent. This solves the problem of representation of negative exponent.

Example: Represent 1259.12510 in single and double precision format.

Solution:

Step 1: Convert decimal number in binary format



$(1259)_{10} = (1001110\ 1011)_2$ and $(0.125)_{10} = (0.001)_2$

Binary number = 10011101011 + 0.001 = 10011101011.001

Step 2: Normalize the number

$10011101011.001 = 1.0011101011001 \times 2^{10}$

Step 3: Single precision representation

For a given number S = 0, E = 10 and M = 0011101011001

Bias for single precision format is - 127

E' = E + 127 = 10 + 127 = 13710 = 100010012

Number in single precision format is given as

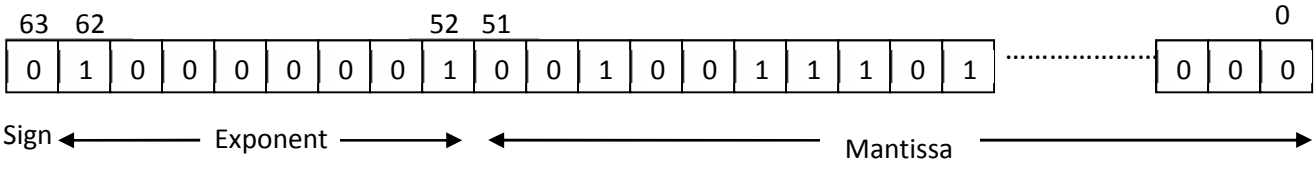
Step 4: Double precision representation. For a given number

S = 0, E = 10, and M = 0011101011001

Bias for double precision format is = 1023

E' = E+1023 = 10+1023 = 103310 = 100000010012

Number in double precision format is given as



Design of Arithmetic Unit:

ALU is responsible to perform the operation in the computer.

The basic operations are implemented in hardware level. ALU is having collection of two types of operations:

Arithmetic Operations (ADD, SUB, MUL, DIV)

Logical Operations (AND, OR, NOT, EX-OR)

Consider an ALU having 4 arithmetic operations and 4 logical operations.

To identify any one of these four logical operations or four arithmetic operations, two control lines are needed. Also to identify the any one of these two groups- arithmetic or logical, another control line is needed. So, with the help of three control lines, any one of these eight operations can be identified.

Consider an ALU is having four arithmetic operations. Addition, subtraction, multiplication and division. Also consider that the ALU is having four logical operations: OR, AND, NOT & EX-OR.

We need three control lines to identify any one of these operations. The input combination of these control lines are shown in Table 4:

Control line C_2 is used to identify the group: logical or arithmetic, i.e. C_2=0: arithmetic operation, C_2=1: logical operation.

Control lines $C_0$and C_1 are used to identify any one of the four operations in a group. One possible combination is given here.

Table 4: Operation Identification in ALU

C_1	C_0	Arithmetic $C_2 = 0$	Logical $C_2 = 1$
0	0	Addition	OR
0	1	Subtraction	AND
1	0	Multiplication	NOT
1	1	Division	**EX-OR**

Figure 10 shows the block diagram of ALU in which the ALU has got two input registers named as A and B and one output storage register, named as C. It performs the operation as:

C = A op B

The input data are stored in A and B, and according to the operation specified in the control lines, the ALU perform the operation and put the result in register C.

As for example, if the contents of controls lines are, 000, then the decoder enables the addition operation and it activates the adder circuit and the addition operation is performed on the data that are available in storage register A and B. After the completion of the operation, the result is stored in register C.

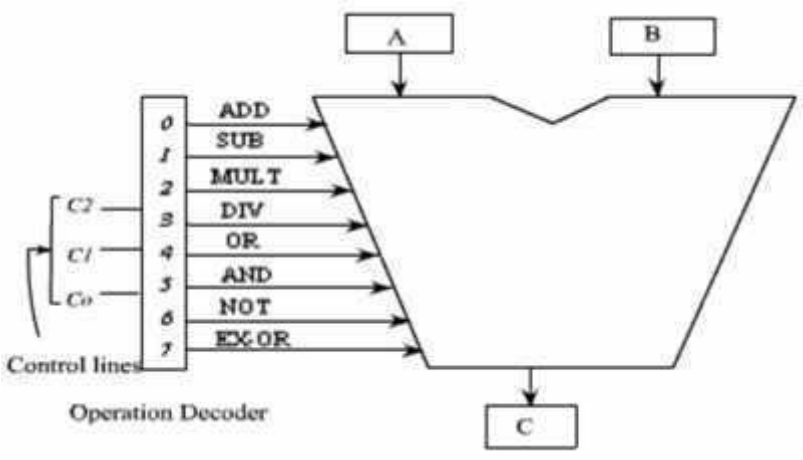


Figure 10: Block Diagram of ALU Design

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in